DEFENCE RESEARCH AGENCY
MALVERN

MEMORANDUM No. 4630

THE DYNAMIC SEMANTICS OF KERNEL ELLA

Author: M G Hill

DEFENCE RESEARCH AGENCY,
MALVERN,
WORCS.

CONDITIONS OF RELEASE

0134070 313436

************************* DRIC U

************************* DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

DEFENCE RESEARCH AGENCY
MALVERN

MEMORANDUM 4630

Title: THE DYNAMIC SEMANTICS OF KERNEL ELLA

Author: M G Hill

Date: August 1992

Summary

This document describes the dynamic semantics of the Kernel of ELLA. The Kernel is a set of data structures into which any ELLA circuit can be transformed. The semantics of two simple languages are explored in order to demonstrate the implementability of the approach undertaken. Correspondence between this work and former analysis is shown.

Intentionally Blank

Contents

Intentionally Blank

1 Introduction

In this memorandum we give a formal definition of the dynamic semantics of the **Kernel** of ELLA. The **Kernel** is a set of data structures defined in [MH91] into which any ELLA description can be transformed. In [HM92] a set of transformational rules are given which have been used to define the implementation of a compiler from Core ELLA to the **Kernel**. Core ELLA represents the heart of the ELLA language and consists of those constructs into which any ELLA description can be transformed via the commercial ELLA system. For a complete description of ELLA the reader is referred to [Com90].

In this memorandum we start by looking at a subset of the **Kernel**, which we shall call K1, which is equivalent to a previously defined language, called $\mathcal{L}$, described by Davies in [Dav88]. We demonstrate how the semantics of this language can be given by the definition of an evaluator. We then proceed to extend K1 to allow function declarations, this is done by introducing the concept of an environment. The definition and application of an evaluator for describing the semantics of a language with an environment are given.

Having demonstrated the evaluation process for these two simple languages we then extend the process to describe the dynamic semantics of the **Kernel**. Throughout this memorandum use will be made of the VDM notation (see [Jon90]).

2 Language K1

2.1 Language Definition

The language K1 will be defined to be that subset of the **Kernel** which encompasses the language constructs defined in [Dav88]. We begin by restating the language $\mathcal{L}$ specified in [Dav88], which is

variables	v	$\in$	E	$\forall\, v \in \mathrm{V}$	
constant	c	$\in$	E	$\forall\, c \in \mathrm{C}$	
pair	(α, β)	$\in$	E	$\forall\, \alpha, \beta \in \mathrm{E}$	
delay	$\Delta_s\, \alpha$	$\in$	E	$\forall\, \alpha \in \mathrm{E}$	
case	$\Box\, \alpha\!: A$	$\in$	E	$\forall\, \alpha \in \mathrm{E}$	$A = \{(\mathrm{c,e})\}$
recure	$\mu v.\alpha$	$\in$	E	$\forall\, v \in \mathrm{V}, \forall\, \alpha \in \mathrm{E}$	

where E represents expressions, C constant values and V variable names. The Δ_s feature allows timing to be incorporated within the language. The Δ_s is taken to be a unit delay, with initial value s, i.e. it passes its input to its output after one time step, with the value at time zero being s. The case expression, $\Box\, \alpha\!: A$, is a multiplexer-like construct which selects its output expression from A, based on the dynamic value of its input α. The set 'A' comprises the complete set of tests and expression-results for the Case construct. The language can describe recursive expressions by means of the last construct, for example a case statement which toggles its input value and which feeds its output into its input via a delay, is represented as (see figure 1)

$$\mu b.\Delta_t \Box b\!: (\{(t,f),(f,t)\})$$

Having stated the Language as defined in [Dav88] we now rewrite this language using a recursive-let style to show the connection with the **Kernel**, and we will call this language K1. We begin by giving the types which can be used in K1.

Types :-

$$\begin{array}{lll} T & \subseteq & \{ \text{(typename, [enum]}^*\text{)} \} \\ Time & \subseteq & \mathbb{N} \end{array}$$

where T represents a basic enumerated type and *Time* will be used to represent evaluation time, analogous to simulator time steps in the full ELLA system.
The language classes which are needed for K1 are the following
Variables, (v ∈ Var) given by:-

$$\text{v} \;::=\; \text{id}$$

where 'id' is an identifier by which the variable is known,

Constants, (c ∈ Const) given by:-

$$\text{c} \;::=\; \text{enum}$$

where 'enum' is a basic enumerated value from a type declaration

Expressions, (e ∈ Expr) given by:-

$$\text{e} \;::=\; \text{v} \mid \text{c} \mid (\text{e}_1, \text{e}_2) \mid Delay_c(\text{e}) \mid Case\ (\text{e}_c, [(\text{c}, \text{e})]^*) \mid \textbf{let}\ \text{d}\ \ \textbf{in}\ \text{e}$$

and Declarations, (d ∈ Decl) given by:-

$$\text{d} \;::=\; \text{v} = \text{e}$$

where in declarations the expression e could contain references to v. The $Delay_c$(e) is a unit delay with initial value 'c' and input expression 'e'. The *Case* (e_c, $[(\text{c}, \text{e})]^*$) is a multiplexer-like construct which delivers the expression part, 'e', of one of the tuples in the sequence, this choice being made by the correspondence between 'c' and the dynamic evaluation of its input 'e_c'.

2.2 Language Evaluation

In order to be able to define the evaluation of expressions in K1 we first introduce the idea of signal history, by means of the following data structure. This data structure will allow dynamic signal values to be collected for each evaluation time step:-

$$\begin{array}{lrcl} History :: & name & : & Var \\ & const & : & Const \\ & time & : & Time \end{array}$$

The sequence $History^*$ will be the set which contains a constant assignment to each variable at every time unit.

The use of *History* means that the dynamic semantics of K1 can be defined by means of the following evaluation function

$$\textit{Evaluate-Exp}: \textit{Expr} \times \textit{History}^* \times \textit{Time} \rightarrow \textit{Const}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\textit{Evaluate-Exp}(e, \textit{history}, t) \triangleq} \\
\multicolumn{2}{l}{\quad \textsf{cases } e \textsf{ of}} \\
\quad v & \rightarrow \textsf{let } val = \iota(i \in \textsf{inds } \textit{history}) \cdot \textit{history}[i] = (v, _, t) \textsf{ in} \\
 & \quad val[2] \\
\quad c & \rightarrow c \\
\quad (e_1, e_2) & \rightarrow (\textit{Evaluate-Exp}(e_1, \textit{history}, t), \textit{Evaluate-Exp}(e_2, \textit{history}, t)) \\
\quad \textit{Delay}_c(e) & \rightarrow \textsf{if } t > 0 \\
 & \quad \textsf{then } \textit{Evaluate-Exp}(e, \textit{history}, t\text{-}1) \\
 & \quad \textsf{else } c \\
\quad \textit{Case}(e_c, [(c, e)]^*) & \rightarrow \textsf{let } \textit{Evaluate-Exp}(e_c, \textit{history}, t) = c_i \textsf{ in} \\
 & \quad \textsf{let } (c_i, e_i) \in [(c, e)]^* \textsf{ in} \\
 & \quad \textit{Evaluate-Exp}(e_i, \textit{history}, t) \\
\quad \textsf{let } d \textsf{ in } e & \rightarrow \textsf{let } \textit{history}^* = \textit{Update-History}(d, \textit{history}, t) \textsf{ in} \\
 & \quad \textit{Evaluate-Exp}(e, \textit{history}^*, t) \\
 & \\
\multicolumn{2}{l}{\quad \textsf{end}}
\end{array}$$

where *val* in the first alternative represents the unique element in the history with the correct

identifer and time. When a let declaration is being evaluated its result could depend on values from previous times. Thus in order to obtain the latest value of the expression a list of relevant history values must be known. The function *Update-History* calculates these and hence delivers a sequence of *History* values of all possible identifiers from the previous time value back to time zero. It does this by calculating the value of the let expression at all previous times. At each time step the history sequence delivered will contain all necessary variable values for the calculation of the value of the expression. Once all previous time calculations have been performed the current value of the expression can be evaluated. The function *Update-History* is given by

$$\textit{Update-History}: \textit{Decl} \times \textit{History}^* \times \textit{Time} \rightarrow \textit{History}^*$$

$$\begin{array}{l}
\textit{Update-History}(d, \textit{history}, t) \triangleq \\
\quad \textsf{cases } d \textsf{ of} \\
\quad v := e \rightarrow \textsf{let } \textit{history}^{\text{-}1} = \textit{history} \textsf{ in} \\
\qquad\qquad \textsf{let } \textit{history}^{i} = \textit{history}^{i\text{-}1} \frown [(v, \textit{Evaluate-Exp}(e, \textit{history}^{i\text{-}1}, i), i)], \quad \textsf{for } i = (t\text{-}1)..0 \textsf{ in} \\
\qquad\qquad \textit{history}^{t\text{-}1} \frown [(v, \textit{Evaluate-Exp}(e, \textit{history}^{t\text{-}1}, t), t)] \\
 \\
\quad \textsf{end}
\end{array}$$

Within the **Kernel** functions are described as collections of signal declarations with the function body being an expression which, possibly, contains references to the signal declarations. If we restrict ourselves to only one function in a **Kernel** environment then the evaluation of that function is equivalent to the evaluation of the functions body expression. Expressions in K1 can therefore be thought of as function body expressions with the let construct representing the signal declarations. In the section 3 we extend K1 by introducing the essence of the **Kernel** environment and demonstrate how the evaluator defined in this section can be enhanced to describe its semantics.

However before proceeding we give a simple example of the evaluation of a function in K1.

2.3 Example

In this example we consider the expression which is shown pictorially in figure 1, and is equivalent to the expression given in section 2.1. In figure 1 the triangular object represents a unit delay, and the Case statement is represented by a box, with the details of the Case statement not shown.

Let b have type (bool, [h,l]) and define the expression as follows

$$exp \equiv \text{let } b = Delay_l(Case(b, [(h,l),(l,h)])) \text{ in } b$$

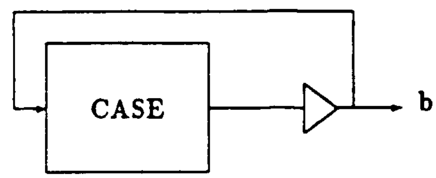

Figure 1: Simple Recursive Example

Now consider the evaluation of the expression at time = 1, with an initial history set to empty e.g. $history = []$, this is possible since there is no direct input to the expression. In this example we will use the shorthand notation of '...' to represent hidden text.

$$\begin{aligned} &\textit{Evaluate-Exp}(exp, [], 1) = \text{let } history^{*} = \textit{Update-History}(b = Delay_l(\ldots), [], 1) \text{ in} \\ &\qquad \textit{Evaluate-Exp}(b, history^{*}, 1) \end{aligned}$$

where

$$\begin{aligned} &\textit{Update-History}(b = Delay_l(\ldots), [], 1) \\ &\quad = \text{let } history^{-1} = [] \text{ in} \\ &\quad\ \ \text{let } history^{0} = [(b, \textit{Evaluate-Exp}(Delay_l(\ldots), [], 0), 0] \text{ in} \\ &\quad\ \ history^{0} \frown [(b, \textit{Evaluate-Exp}(Delay_l(\ldots), history^{0}, 1), 1] \end{aligned}$$

and

$$\begin{aligned} &\textit{Evaluate-Exp}(Delay_l(\ldots), [], 0) = l \\ &\textit{Evaluate-Exp}(Delay_l(\ldots), [(b, l, 0)], 1) \\ &\quad = \textit{Evaluate-Exp}(Case(b, [(h,l),(l,h)]), [(b,l,0)], 0) \\ &\quad = h \end{aligned}$$

thus

$$\textit{Update-History}(b = Delay_l(\ldots), [], 1) = [(b,l,0),(b,h,1)]$$

and hence

$$\textit{Evaluate-Exp}(exp, [], 1) = \textit{Evaluate-Exp}(b, [(b,l,0),(b,h,1)], 1) = h$$

Similarly it can be shown that

$$\textit{Evaluate-Exp}(exp, [], 2) = l$$
$$\textit{Evaluate-Exp}(exp, [], 3) = h \text{ ...etc}$$

Notice that since the expression has no direct inputs it is possible to start any evaluation with an empty history, the history of internal signals always get built up as the evaluation process progresses.

3 Language K2

3.1 Language Definition

We now proceed to define the language K2, which is a direct extension of K1, that introduces an environment with function declarations. The main evaluation function is called *Evaluate-Fn* and this evaluates the result of a function given a specified input history. This model of evaluation is 'backward-looking' in the sense that the value of the present output is calculated from the dependency of values from a previous time.

The language classes have been changed from K1 to
Expressions, (e $\in$ Expr) :-

$$e \quad ::= \quad \text{Sig(no)} \mid c \mid (e_1, e_2) \mid \text{Delay}_c(e) \mid \text{Case}(e_{ch}, [(c, e)]^*) \mid \text{Call(fnno, e)} \mid \text{Index(e,ind)}$$

Constants, (c $\in$ Const) :-

$$c \quad ::= \quad \text{enum} \mid (c_1, c_2)$$

where 'Sig(no)' is defined tc be the result delivered by the no^{th} signal declaration of the enclosing function declaration (see below). Function calls are all implicit and are represented by 'Call(fnno, e)' which is the instantiation of a function which is the $fnno^{th}$ declaration in the environment with 'e' assigned to its input expression. Tuples and indexing have been included within K2 to demonstrate how structured function inputs can be handled.

An environment which collects together all the type and function declarations is given by

Env :: *typedec* : *Typedec**
fndec : *Fndec**

with

Typedec :: *typename* : *Id*
enum : *Enum**

Fndec :: *fnname* : *Id*
signal : *Signal**
expr : *Expr*

Signal :: *name* : *Id*
expr : *Expr*

where a function declaration is defined to have three fields. The first field is the name of the function, the second is a sequence of signal declarations which closely correspond to the 'let *signalname* in *expression*' of K1. The last field is the expression delivered by the function. Signal declarations in a function are referenced by their position in the signal declaration field of the function declaration. Thus $sig(3)$ represents the result of the third signal declaration.

3.2 Language Evaluation

In a similar manner to K1 we need to define a history-like data structure in order to hold function input data history. The *History* concept of K1 can however be simplified for K2 by means of the introduction of an *Input* data structure as defined by

$$\begin{array}{ll} Input :: & const : Const \\ & time : \mathbb{N} \end{array}$$

Here we see that, compared with *History*, an identifier field is no longer necessary. This is possible since *Input* is defined to be a sequence of values (over time) which are the inputs to a function. Thus all function inputs will be represented within *Input* as having only one input terminal. Naturally some function definitions may contain inputs which are structures of named types. In this case within the function signal field the first element will be defined as (_input,) and then named input terminals will refer to this field. For example if a function signal field contained the following entry (input1, sig(1)), it would imply that the function had one named input, called 'input1'. For functions with structured inputs the following could be possible within the function signal field (input2, Index(sig(1),2)), which would mean that 'input2' was the second input terminal of the function. For this language we shall limit the size of a structure to two items, this restriction will be removed in the next section.

As in the case of K1 some expressions could need the results of expressions from previous time steps. Thus to evaluate each function a sequence of *Inputs* will be needed. These sequences define collections of input values to the function from time 0 to the current time i.e. $input = [(c_t, time_t), ..., (c_0, time_0)]$. By using a model of evaluation which calculates the present function value by recourse to all previous function inputs, the storing of the values of signals internal to a function declaration is avoided.

Since the structure *Input* has replaced *History* the function *Update-History* of K1 needs to be replaced by a function *Update-Inputs*. This function performs a similar operation to *Update-History*, but instead of calculating all possible values of each variable it calculates all values to the input of the function from previous times. This naturally means that in order to evaluate a function, F say, at some time, t say, the evaluation of F will need to know the complete history of all inputs to F from time zero to time t. Any internal function calls within F can calculate the history of inputs to those functions by knowing the history of inputs to F. In such cases, as can be seen by *Evaluate-Exp*, a function call needs to generate a history of all local inputs to that function call.

Thus the function for generating a sequence of input values (over time) is defined as

$$\textit{Update-Inputs}: \textit{Expr} \times \textit{Signal}^* \times \textit{Input}^* \times \textit{Time} \to \textit{Input}^*$$

$$\begin{array}{l} \textit{Update-Inputs}(e, signals, inputs, t) \triangleq \\ \quad \textbf{if } t \geq 0 \\ \quad \textbf{then let } inputs' = \textit{Update-Inputs}(e, signals, inputs, (t\text{-}1)) \textbf{ in} \\ \qquad inputs' \dagger [(\textit{Evaluate-Exp}(e, signals, inputs', t), t)] \\ \quad \textbf{else } inputs \end{array}$$

Concatenation in the definition of *Update-History* in K1 has been replaced by overwrite for *Update-Inputs*. This is chosen since each function references its input by the tag *_input*. Thus two different function calls would both reference their respective inputs by this tag. By using the notion of overwriting of *Input* values we can avoid the stacking and unstacking of *Input* sequences. This is of course only possible since each *Input* sequence is local to a function call, and there are no local scoping rules.

Evaluation of an expression has a similar format to that for K1. The following function defines the evaluation for each expression in K2.

$Evaluate\text{-}Exp: Expr \times Signal^* \times Input^* \times Time \rightarrow Const$

$Evaluate\text{-}Exp(e, signals, inputs, t) \triangleq$

$$
\begin{array}{ll}
\textsf{cases } e \textsf{ of} & \\
sig(no) & \rightarrow \textsf{let } signal = signals[no] \textsf{ in} \\
& \quad \textsf{if } signal.name = \text{"\_input"} \\
& \quad \textsf{then let } val = \iota\,(i \in \textsf{inds}\, inputs) \cdot inputs[i] = (\_, t) \textsf{ in} \\
& \quad\quad val[1] \\
& \quad \textsf{else } Evaluate\text{-}Exp(signal.expr, signals, inputs, t) \\
c & \rightarrow c \\
(e_1, e_2) & \rightarrow (Evaluate\text{-}Exp(e_1, signals, inputs, t), \\
& \quad Evaluate\text{-}Exp(e_2, signals, inputs, t)) \\
Delay_c(e) & \rightarrow \textsf{if } t > 0 \\
& \quad \textsf{then } Evaluate\text{-}Exp(e, signals, inputs, t\text{-}1) \\
& \quad \textsf{else } c \\
Case(e_c, [(c, e)]^*) & \rightarrow \textsf{let } Evaluate\text{-}Exp(e_c, signals, inputs, t) = c_i \textsf{ in} \\
& \quad \textsf{let } (c_i, e_i) \in [(c, e)]^* \textsf{ in} \\
& \quad Evaluate\text{-}Exp(e_i, signals, inputs, t) \\
Call(fnno, e) & \rightarrow \textsf{let } inputs' = Update\text{-}Inputs(e, signals, inputs, t) \textsf{ in} \\
& \quad Evaluate\text{-}Fn(fnno, inputs', t) \\
Index(e, ind) & \rightarrow \textsf{let } (c_1, c_2) = Evaluate\text{-}Exp(e, signals, inputs, t) \textsf{ in} \\
& \quad \textsf{if } ind = 1 \\
& \quad \textsf{then } c_1 \\
& \quad \textsf{else } c_2 \\
\textsf{end} &
\end{array}
$$

where *signal.name* is the name field of the signal data structure, and *val* is the unique input element with the desired value of time.

The above two functions can now be combined into a function for evaluating the $fnno^{th}$ function in an environment, given a specific input history list, at a specific time as

$Evaluate\text{-}Fn: Fnname \times Input^* \times Time \rightarrow Const$

$Evaluate\text{-}Fn(fnno, inputlist, time) \triangleq$
  $\textsf{let } fdec = (EnvFndec)[fnno] \textsf{ in}$
  $Evaluate\text{-}Exp(fdec.expr, fdec.signal, inputlist, time)$

where *env* is the complete environment containing the function, EnvFndec the operator which returns the function declaration field of the environment, and index [fnno] the $fnno^{th}$ element of the resulting sequence (a glossary of the symbols used in this document is given in appendix A).

In order to show the evaluation process we now give two examples. The first example is a repeat of that given in section 2.3, the second example is of a reset/set flip-flop.

3.3 Example 1

Here we rewrite the example of the section 2.3 as two functions, called A and B, to demonstrate evaluation in K2. The environment is given by

```
Env  =  (  [(bool, [h,l])],
           [(A, [(_input, _)], Case(sig(1), [(h,l),(l,h)])),
            (B, [(_input, _), (b, Delay_l(Call(1,sig(2))))], sig(2)]
        )
```

This environment can be written more readably as

```
Type bool = (h | l).

Fn A = (bool:_input) -> bool:
CASE _input OF
   h: l,
   l: h
ESAC.

Fn B = (bool:_input) -> bool:
( Let b = Delay{l} (A b)
  In b
).
```

It should be noted that this format does not conform to any particular language, its purpose is to give a visualisation of the above environment where the calls of signals can be more readily seen.

Now consider the Evaluation of function B at time '1' with signal inputs set to $initial = [(h,1),(h,0)]$. In this case we start with a non-empty input history in order to show the overwriting of input values. In the following the syntax '...' is used to signify that parts of the expression have been left out in order to aid readability.

$$\begin{aligned}
\textit{Evaluate-Fn}(2, initial, 1) &= \textit{Evaluate-Exp}(sig(2), [(b, Delay(...))], initial, 1)\\
&= \textit{Evaluate-Exp}(Delay(...), [(b, ...)], initial, 1)\\
&= \textit{Evaluate-Exp}(Call(1, sig(2)), [(b, ...)], initial, 0)\\
&= \text{let } i' = \textit{Update-Input}(sig(2), [(b, ...)], initial, 0) \text{ in}\\
&\quad\ \textit{Evaluate-Fn}(1, i', 0)
\end{aligned}$$

where

$$\begin{aligned}
&\textit{Update-Input}(sig(2), [(b...)], initial, 0) =\\
&\qquad \text{let } i' = initial \text{ in}\\
&\qquad i' \dagger [(\textit{Evaluate-Exp}(sig(2), [(b...)], i', 0), 0]\\
&\qquad = i' \dagger [(\textit{Evaluate-Exp}(Delay(...), i', 0), 0]\\
&\qquad = i' \dagger [(l, 0)]
\end{aligned}$$

therefore

$$\begin{aligned}
\textit{Evaluate-Fn}(2, \textit{initial}, 1) &= \textsf{let } i' = \textit{initial} \dagger [(l,0)] = [(h,1),(l,0)] \textsf{ in} \\
&\quad \textit{Evaluate-Fn}(1, i', 0) \\
&= \textit{Evaluate-Exp}(\textit{Case}(\ldots), [(_input, _)], i', 0) \\
&= \textsf{let } cc = \textit{Evaluate-Exp}(\textit{sig}(1), [(_input, _)], i', 0) = l \textsf{ in} \\
&\quad \textsf{let } (cc, e) \in [(h,l),(l,h)] \textsf{ in} \\
&\qquad \textit{Evaluate-Exp}(e) \\
&= h
\end{aligned}$$

and hence the result of evaluating B is the same as for evaluating the expression in the previous section.

3.4 Example 2

In this example we demonstrate the result of evaluating of an RS Flip Flop. The definition of an appropriate environment is given by

```
env = ( [(bool, [h,l])]
        [(NOR, [(_input, _)], Case(sig(1), [((l,l),h),((l,h),l),((h,l),l),((h,h),l)]))
         (RSFF, [ (_input, _),
                  (in1, Index(sig(1), 1)),
                  (in2, Index(sig(1), 2)),
                  (del1, Delay_l(sig(6))),
                  (del2, Delay_l(sig(7))),
                  (nor1, Call(1, (sig(2), sig(5)))),
                  (nor2, Call(1, (sig(3), sig(4)))),
                  ],
                  sig(4))
        ]
      )
```

It can be noted that the function RSFF has an input which is a structure of two items called in1 and in2. There are two function calls to NOR the outputs of which are named nor1 and nor2, with the result of RSFF being the expression 'sig(4)'. This circuit is shown in figure 2.

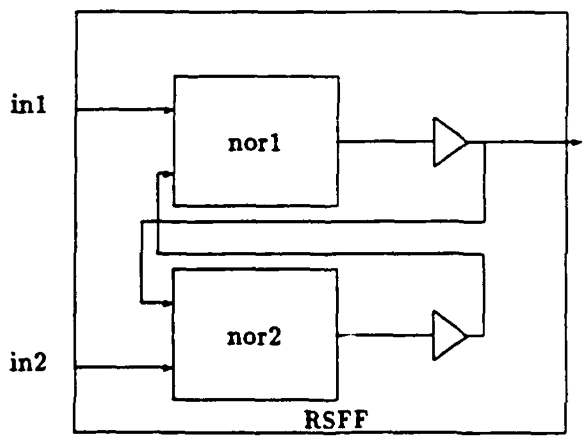

Figure 2: RSFF

When this circuit is evaluated for time t = 3, say, an input list of the form [((l,h),0), ((l,h),1), ((l,h),2), ((l,h),3)] is needed. Evaluation for this case gives

$$\begin{aligned}&\textit{Evaluate-Fn}(2, [((l,h),0),((l,h),1),((l,h),2),((l,h),3)],3)\\ &\quad = \textit{Evaluate-Exp}(sig(4), [(\_input,\_),...,(nor2,...)], [((l,h),0)...((l,h),3)],3)\\ &\quad = ...\\ &\quad = h\end{aligned}$$

where the result has been obtained from a computer implementation of the evaluation functions defined in this section. The result of applying *Evaluate-Exp* at the outer level at time t=3 is to 'unwrap', or fold back, the circuit in time. The result of this is most readily seen in pictorial form and a representation is given in figure 3.

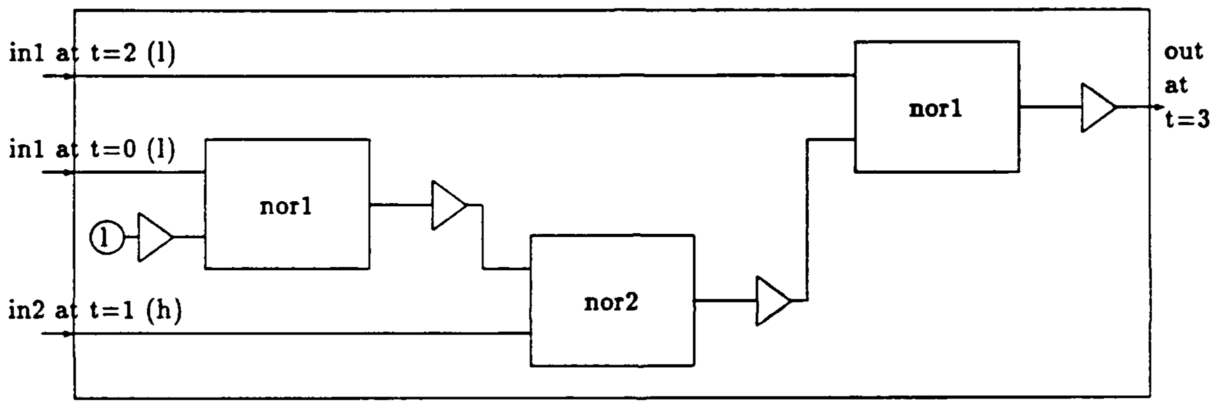

Figure 3: Time Expanded RSFF at time t = 3

4 The Semantics of the Kernel

4.1 Kernel verses K2

We now proceed to give the definition of the **Kernels** dynamic semantics. The **Kernel** is really an extended version of K2 where the following extensions have been made (the names under the **Kernel** column refer to **Kernel** data structures, see appendix B)

K2		Kernel
expression	→	Unit
constant	→	Const
environment	→	Env
typedec	→	Typedec
fndec	→	Fndec
signal	→	Signaldec

Some of the structures in the **Kernel**, like signaldec, have fields which hold typing information. This is not strictly necessary for the definition of the dynamic semantics but we leave the

structures as defined in [MH91] for completeness. The *Input* history of a function declaration remains unaltered. The evaluation of a **Kernel** function will follow the same process as that of a K2 function. However, due to the large increase of terms in going from expressions in K2 to units in the **Kernel**, before we can define the *Evaluate-Fn* function for the **Kernel** a number of other functions are needed, these are defined in the subsequent sections.

4.2 Environment Access Operators

A **Kernel** environment contains all the Type and Function declarations, see [MH91], and the operators in this section define how declarations are accessed.

Extracting a Type declaration:-

EnvTypedec()*tydec*: *kTypedec*
ext rd *env*: *Env*
post *tydec* = (*env.typedec*)

Extracting a Function declaration:-

EnvFndec()*fndec*: *kFndec*
ext rd *env*: *Env*
post *fndec* = (*env.fndec*)

Since the environment is not going to be altered by the evaluation of a function there is no necessity for operators which update the environment (the environment only gets updated during compilation, see [HM92] for a description of the operators necessary for compilation).

4.3 General Functions

In this section we define functions which are general to the evaluation process. We start by looking at two functions which deliver the type of an expression. The first delivers the type of a constant expression and is given by

Type-Of-Const : *kConst* → *kType*

Type-Of-Const(*const*) $\triangleq$
cases *const* of
enum(*typeno*, _) → **typeno**(*typeno*)
string(*typeno*, [$tag_1, \cdots, tag_n$]) → **stringtype**(n, **typeno**(*typeno*))
conststring(*size*, *c*) → **stringtype**(*size*, *Type-Of-Const*(*c*))
consts([$c_1, \cdots, c_n$]) → **types**([*Type-Of-Const* (c_1), $\cdots$, *Type-Of-Const* (c_n)])
constassoc(**enum**(*typeno*, _), _) → **typeno**(*typeno*)
constquery(*type*) → *type*
constvoid → **typevoid**
end

The type of a unit expression can be found by means of the following function. For some of the unit expressions, e.g. **extract**, the required typing information is not held directly in the unit data structure. In such cases the information needs to be located within the environment. For

the case of **extract** this means finding the correct type declaration, selecting the sequence of enumerated values of the type, locating the desired element of that sequence and then finding the type of the optional part, which will only be a valid type in the case of an associated value.

Type-Of-Unit: $kUnit \times Signaldec^* \rightarrow kType$

Type-Of-Unit$(unit, sigdecs)$ $\underline{\triangle}$

cases *unit* of

enumerated	$\rightarrow$ *Type-Of-Const* (*enumerated*)
conc(_, _, *type*)	$\rightarrow$ *type*
instance(*fnno*, _)	$\rightarrow$ (*EnvFndec*)[*fnno*].*outputtype*
unitassoc(**enum**(*typeno*, _), _)	$\rightarrow$ **typeno**(*typeno*)
extract(**enum**(*typeno*, *tagno*), _)	$\rightarrow$ ((*EnvTypedec*)[*typeno*].*new*)[*tagno*].*typeopt*
signal(*signalno*)	$\rightarrow$ *sigdecs*[*signalno*].*type*
index(_, _, *outputtype*)	$\rightarrow$ *outputtype*
trim(_, _, _, *outputtype*)	$\rightarrow$ *outputtype*
dyindex(_, _, *outputtype*)	$\rightarrow$ *outputtype*
replace(*u*, _, _)	$\rightarrow$ *Type-Of-Unit*(*u*, *sigdecs*)
unitquery(*type*)	$\rightarrow$ *type*
unitvoid	$\rightarrow$ **typevoid**
unitstring(*size*, *u*)	$\rightarrow$ **stringtype**(*size*, *Type-Of-Unit* (*u*, *sigdecs*))
caseclause(_, _, *u*)	$\rightarrow$ *Type-Of-Unit* (*u*, *sigdecs*)
units($[u_1, \cdots, u_n]$)	$\rightarrow$ **types**([*Type-Of-Unit* (u_1, *sigdecs*), $\cdots$, *Type-Of-Unit* (u_n, *sigdecs*)])

end

In some cases when handling constant strings it is easier to convert them into enumerated strings. Although this removes replication information it simplifies the evaluation process and is therefore desirable. The following function defines the conversion

Conv-String: $kConst \rightarrow kConst$

Conv-String(c) $\underline{\triangle}$

cases *c* of

string(_, _)	$\rightarrow$ *c*
conststring(*size*, **enum**(*typeno*, *tagno*))	$\rightarrow$ **string**(*typeno*, [$tagno^{size}$])

others **constquery**(*Type-Of-Const*(*c*))

end

Within the **Kernel** there are a number of Built-in Operators which are optimal in their handling of ambiguity. The following function checks that the biop name *nm* is one of them.

Optimal-Biop: $Biopname \rightarrow \mathbf{B}$

Optimal-Biop(nm) $\underline{\triangle}$

$(nm = EQ\_US) \vee (nm = GT\_US) \vee (nm = GE\_US) \vee (nm = LT\_US) \vee (nm = LE\_US)$
$\vee (nm = EQ\_S) \vee (nm = GT\_S) \vee (nm = GE\_S) \vee (nm = LT\_S) \vee (nm = LE\_S)$

Within the **Kernel** it is possible to have an aliased type. Before any checks on the type can be performed the alias name must be removed by means of the following function

Get-Type : *kType* $\rightarrow$ *kType*

Get-Type(*ty*) $\triangleq$
  cases *ty* **of**
  types([*ktype*$_1$, $\cdots$, *ktype*$_k$]) $\rightarrow$ **types**([*Get-Type*(*ktype*$_1$), $\cdots$, *Get-Type*(*ktype*$_k$)]),
  typename(_, *ktype*) $\rightarrow$ *Get-Type*(*ktype*)
  stringtype(*size*, *ktype*) $\rightarrow$ **stringtype**(*size*, *Get-Type*(*ktype*))
  others *ty*

  end

The equality of two types can be confirmed by the following

Type-Equals : *kType* $\times$ *kType* $\rightarrow$ **B**

Type-Equals(ty_1, ty_2) $\triangleq$
  cases (*Get-Type*(ty_1), *Get-Type*(ty_2)) **of**
  (**typeno**($typeno_1$), **typeno**($typeno_2$)) $\rightarrow$ ($typeno_1 = typeno_2$)
  (**stringtype**(s_1, tn_1), **stringtype**(s_2, tn_2)) $\rightarrow$ ($s_1 = s_2$) $\wedge$ *Type-Equals*(tn_1, tn_2)
  (**types**([t_1, ..., t_k]), **types**([s_1, ..., s_j])) $\rightarrow$ $j = k \bigwedge_{i=1..k}$ *Type-Equals*(t_i, s_i)
  (**typevoid**, **typevoid**) $\rightarrow$ **true**
  others false

  end

In some expressions the ELLA unknown, or query, value can be passed into its input. In certain cases, e.g. RAM, this needs to be checked for so that an appropriate query value can be returned. The following function checks to see whether a constant expression contains any part which has a query value.

Has-Query: *kConst* $\rightarrow$ **B**

Has-Query(*c*) $\triangleq$
  cases *c* **of**
  consts([c_1, $\cdots$, c_k]) $\rightarrow$ $\bigvee_{i=\{1..k\}}$ *Has-Query*(c_i)
  conststring(_, *c*) $\rightarrow$ *Has-Query*(*c*)
  constassoc(_, *c*) $\rightarrow$ *Has-Query*(*c*)
  constquery(_) $\rightarrow$ **true**
  others false

  end

ELLA integers are tagged integers where the value of an integer signal is represented as an enumerated data structure with a tag number offset from the types lower bound value. Thus some constructs, e.g. dynamic indexing, will need to know the range of an integer signal so that the appropriate offset can be used. The following function looks in the environment type declarations and returns the type information for the integer in question

$Find\text{-}Integer\text{-}Type$: $kType \rightarrow Tagname \times Lowerbound \times Upperbound$

$Find\text{-}Integer\text{-}Type(ktype) \triangleq$
let $\mathbf{typeno}(typeno) = Get\text{-}Type(ktype)$ in
cases $(EnvTypedec)[typeno]$ of
$\mathbf{typedec}(\_, \mathbf{ellaint}(t, l, u)) \rightarrow t,\ l,\ u$

end

In some functions, eg. BIOPs, it is necessary to ensure that the input types supplied are from a two valued enumerated type. This is particularly important for bit string operations where the type must be a two valued character type. It can be noted that two valued ELLA integers are not allowed in the BIOPs and hence they need not be checked for by this function. The complete function is therefore

$Check\text{-}Two\text{-}Val : kType \rightarrow \mathbf{B}$

$Check\text{-}Two\text{-}Val(ty) \triangleq$
let $\mathbf{typeno}(typeno) = Get\text{-}Type(ty)$ in
cases $(EnvTypedec)[typeno].new$ of
$\mathbf{tags}(TagSeq) \rightarrow \mathrm{len}\,(TagSeq) = 2$
$\mathbf{chars}(Charseq) \rightarrow \mathrm{len}\,(CharSeq) = 2$

end

The REFORM function reforms one type structure into another. In order for this to be allowed the types must flatten to the same basic structure. The following function takes a structured constant expression and flattens it to its lowest form. The REFORM evaluation function will then build up the new type from this flattened form, via comparison with the desired output type.

$Flatten\text{-}const$: $kConst \rightarrow kConst^*$

$Flatten\text{-}const(c) \triangleq$
cases c of
$\mathbf{consts}([c_1, \cdots, c_k]) \rightarrow Flatten\text{-}Const(c_1) \frown \cdots \frown Flatten\text{-}Const(c_k)$
others $[c]$

end

The following function, which converts a constant expression into a unit expression, will be used by the evaluation function for the built in function bodies at the outer level. This is needed to reduce the necessity for two different evaluation functions for the built in function bodies. Namely, one for function calls with unit inputs and one for outer function instances with constant inputs. Since the basic evaluation for the built in functions would be equivalent in both cases we can combine their calling function by using this convert function.

Convert-Const-Unit : *kConst* $\rightarrow$ *kUnit*

Convert-Const-Unit(*c*) $\triangleq$
cases *c* of
conststring(*size*, c_1) $\rightarrow$ **unitstring**(*size*, *Convert-Const-Unit*(c_1))
consts([$c_1, \cdots, c_k$]) $\rightarrow$ **units**([*Convert-Const-Unit*(c_1), $\cdots$,
Convert-Const-Unit(c_k)])
constassoc(*enum*, c_1) $\rightarrow$ **unitassoc**(*enum*, *Convert-Const-Unit*(c_1))
constquery(*ty*) $\rightarrow$ **unitquery**(*ty*)
constvoid $\rightarrow$ **unitvoid**
others *c*

end

4.4 Unit Evaluation

In this section functions are defined which will be used by the main evaluation function for unit expressions. It can be noted that the static semantics of Core ELLA (see [MH91]) ensures checks on the bounds of arrays. Therefore in the functions defined in this section it will be assumed that such items need no further checking. Some functions will however return the query, or unknown, value for the cases when the signal input is undefined.

The value of a signal declaration is given by the following function. In the **Kernel** an input signal is denoted by the structure **input** in the signal declaration field of a function declaration. Unlike K2 there is no unique input field and hence an appropriate index needs to be generated in the case of structured inputs. In the case of an input signal the value of *val* is the unique input which has the correct 'time' value. Evaluate-Index is then called if a function specification has more than one named input terminal and the appropriate value from the input structure is sort. If a signal is not an input then the function returns the value of its associated expression.

Sig: $\mathbb{N}$ $\times$ *Signaldec** $\times$ *Input** $\times$ *Time* $\rightarrow$ *kConst*

Sig(*signo*, *sigdec*, *inputs*, *time*) $\triangleq$
let *signaldec* = *sigdec*[*signo*] in
if *signaldec.unitorinput* = **input**
then let *val* = ι(*i* $\in$ inds *inputs*) $\cdot$ *inputs*[*i*] = (_, *time*) in
if (len *sigdec* > 1) $\wedge$ *sigdec*[2]*.unitorinput* = **input**
then *Evaluate-Index*(*val*[1], *signo*, *signaldec.type*)
else *val*[1]
else *Evaluate-Unit*(*signaldec.unitorinput*, *sigdec*, *inputs*, *time*)

We now define a function which delivers the value of the extracted part of an associated type

Evaluate-Extract: *kConst* $\times$ *kEnum* $\rightarrow$ *kConst*

Evaluate-Extract(*c*, *enum*) $\triangleq$
cases *c* of
constassoc(*enum*, *const*) $\rightarrow$ *const*
others **constquery**(*Type-Of-Const*(*c*))

end

To obtain the value of a structure which has been indexed we need the following function. Since *ind* is a static index no checking on its value is needed here since this would have occurred at the static semantic stage.

$Evaluate\text{-}Index: kConst \times \mathbb{N} \times kType \rightarrow kConst$

$Evaluate\text{-}Index(c, ind, ty) \triangleq$

cases c of
 $\mathbf{string}(typeno, [tg_1, \cdots, tg_k]) \rightarrow \mathbf{enum}(typeno, tg_{ind})$
 $\mathbf{conststring}(\_, const) \rightarrow const$
 $\mathbf{consts}([c_1, \cdots, c_k]) \rightarrow c_{ind}$
others $\mathbf{constquery}(ty)$

end

To obtain the result of a structure which has been trimmed we have a similar function to that of an index, namely

$Evaluate\text{-}Trim: kConst \times \mathbb{N} \times \mathbb{N} \times kType \rightarrow kConst$

$Evaluate\text{-}Trim(c, ind1, ind2, ty) \triangleq$

cases c of
 $\mathbf{string}(typeno, [tg_1, \cdots, tg_k]) \rightarrow \mathbf{string}(typeno, [tg_{ind1}, \cdots, tg_{ind2}])$
 $\mathbf{conststring}(\_, const) \rightarrow \mathbf{conststring}(ind2\text{-}ind1 + 1, const)$
 $\mathbf{consts}([c_1, \cdots, c_k]) \rightarrow \mathbf{consts}([c_{ind1}, \cdots, c_{ind2}])$
others $\mathbf{constquery}(ty)$

end

To obtain the value of a structure which has been indexed dynamically we first locate the appropriate integer type declaration, calculate what corresponding static index is required and then perform that index. If the signal which is indexing is the query value then a query result is returned.

$Evaluate\text{-}Dyindex: kConst \times kConst \times kType \rightarrow kConst$

$Evaluate\text{-}Dyindex(c_1, c_2, ty) \triangleq$

cases c_2 of
 $\mathbf{constquery}(\_) \rightarrow \mathbf{constquery}(ty)$
others let $\mathbf{enum}(typeno, tagno) = c_2$ in
  let $(\_, lwb, upb) = Find\text{-}Integer\text{-}Type(\ \mathbf{typeno}(typeno))$ in
   $Evaluate\text{-}Index(c_1, lwb + tagno\text{-}1, ty)$

end

The replacing of an element of a structure by a new element whose location is determined by a dynamic index is carried out by the REPLACE primitive. The semantics of REPLACE can be given by the following function

Evaluate-Replace: *kConst* $\times$ *kConst* $\times$ *kConst* $\rightarrow$ *kConst*

```
Evaluate-Replace(c1, c2, c3) ≜
    cases c2 of
     constquery(_) → constquery(ty)
    others let enum(typeno, tagno) = c2 in
          let (_, lwb, upb) = Find-Integer-Type( typeno(typeno)) in
          let ind = lwb + tagno-1 in
            cases c1 of
             string(typeno, [tg1, ···, tgk]) → let c3 = enum(typeno, tag) in
                                               string(typeno, [tg1, ···, tg(ind-1), tag, tg(ind+1), ···, tgk])
             conststring(size, const)       → let const = enum(typeno, tg) in
                                               let string = string(typeno, [tg^size]) in
                                                 Evaluate-Replace(string, c2, c3)
             consts([c1, ···, ck])          → consts([c1, ···, c(ind-1), c3, c(ind+1), ···, ck])
            others constquery( Type-of-Const(c1))

            end

    end
```

Concatenation can be split into two parts, namely strings and structures. The concatenation of structures is defined by

Conc-Const: *kConst* $\times$ *kConst* $\times$ *kType* $\rightarrow$ *kConst*

```
Conc-Const(c1, c2, ty) ≜
    cases c1 of
     consts(cseq1)  → cases c2 of
                       consts(cseq2)  → if Type-Equals( Type-Of-Const(cseq1[1]),
                                                Type-Of-Const(cseq2[1]))
                                        then consts(cseq1 ⁀ cseq2)
                                        else if Type-Equals( Type-Of-Const(cseq1[1]),
                                                     Type-Of-Const(c2))
                                             then consts(cseq1 ⁀ [c2])
                                             else consts([c1] ⁀ cseq2)
                       constquery(_) → constquery(ty)
                      others consts(cseq1 ⁀ [c2])

                      end

     constquery(_) → constquery(ty)
    others consts([c1] ⁀ cseq2)

    end
```

To obtain the value of concatenating two string structures we have

$$Conc\text{-}String: kConst \times kConst \times kType \rightarrow kConst$$

$$
\begin{array}{l}
Conc\text{-}String(c_1, c_2, ty) \underline{\triangle} \\
\quad \textsf{cases}\ (c_1, c_2)\ \textsf{of} \\
\quad (\ \mathbf{string}(ty, [tgl_1, \cdots, tgl_k]) \\
\qquad \mathbf{string}(ty, [tg2_1, \cdots, tg2_m])) \rightarrow \ \mathbf{string}(ty, [tgl_1, \cdots, tgl_k, tg2_1, \cdots, tg2_m]) \\
\quad (\ \mathbf{string}(ty, [tgl_1, \cdots, tgl_k]) \\
\qquad \mathbf{conststring}(\_, \_)) \qquad\qquad \rightarrow Conc\text{-}String(c_1, Conv\text{-}String(c_2)) \\
\quad (\ \mathbf{conststring}(\_, \_), \_) \qquad\qquad\ \rightarrow Conc\text{-}String(Conv\text{-}String(c_1), c_2) \\
\quad \textsf{others}\ \ \mathbf{constquery}(ty) \\
\\
\quad \textsf{end}
\end{array}
$$

The evaluation of a Case statement is defined by the following functions where the result

delivered is the query value whenever the input chooser has an unknown element. The function *Match* compares the chooser value against all alternatives in the Case statement and is defined as

$$Match: kConst \times kConstset \rightarrow \mathbb{B}$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{Match(const, constset) \underline{\triangle}} \\
\multicolumn{2}{l}{\quad \textsf{cases}\ (const, constset)\ \textsf{of}} \\
\quad (\ \mathbf{enum}(\_, tagno_1),\ \ \mathbf{enum}(\_, tagno_2)) & \rightarrow tagno_1 = tagno_2 \\
\quad (\ \mathbf{string}(\_, [tagno_{11}, \cdots, tagno_{1k}]), & \\
\qquad \mathbf{string}(\_, [tagno_{21}, \cdots, tagno_{2k}])) & \rightarrow \bigwedge\limits_{i=\{1..k\}} (tagno_{1_i} = tagno_{2_i}) \\
\quad (\ \mathbf{constassoc}(\ \mathbf{enum}(\_, tagno_1), const_1), & \\
\qquad \mathbf{constsetassoc}(\ \mathbf{enum}(\_, tagno_2), constset_2)) & \rightarrow tagno_1 = tagno_2 \wedge \\
 & \quad Match(const_1,\ constset_2) \\
\quad (\ \mathbf{consts}([c_1, \cdots, c_k]), & \\
\qquad \mathbf{constsets}([cs_1, \cdots, cs_k])) & \rightarrow \bigwedge\limits_{i=\{1..k\}} Match(c_i,\ cs_i) \\
\quad (\_,\ \ \mathbf{constsetalts}([csa_1, \cdots, csa_k])) & \rightarrow \bigvee\limits_{i=\{1..k\}} Match(const,\ csa_i) \\
\quad (\ \mathbf{conststring}(size_a, c_a), & \\
\qquad \mathbf{constsetstring}(size_b, cset_b)) & \rightarrow (size_a = size_b) \wedge Match(c_a,\ cset_b) \\
\quad (\ \mathbf{conststring}(size, c),\ \mathbf{string}(ty, [tg_1, \cdots, tg_k])) & \rightarrow (size = k) \\
 & \quad \bigwedge\limits_{i=\{1..k\}} Match(c,\ \mathbf{enum}(ty, tg_i)) \\
\quad (\ \mathbf{string}(ty, , [tg_1, \cdots, tg_k]), & \\
\qquad \mathbf{constsetstring}(size, cset)) & \rightarrow (size = k) \\
 & \quad \bigwedge\limits_{i=\{1..k\}} Match(\ \mathbf{enum}(ty, tg_i), cset) \\
\quad (\_,\ \ \mathbf{constsetany}(type)) & \rightarrow \textsf{true} \\
\multicolumn{2}{l}{\quad \textsf{others false}} \\
 & \\
\multicolumn{2}{l}{\quad \textsf{end}}
\end{array}
$$

The evaluation of a Case statement is defined as

$$\textit{Evaluate-Case}: \textit{kConst} \times \textit{Case}^{*} \times \textit{kUnit} \times \textit{Signaldec}^{*} \rightarrow \textit{kUnit}$$

$\textit{Evaluate-Case}(\textit{chooser}, \textit{cseq}, \textit{unit}, \textit{sigdecs}) \underline{\triangle}$
  if $\exists(i \in \textsf{inds}\ \textit{cseq}) \cdot \textit{Match}(\textit{chooser}, \textit{cseq}[i].\textit{constset})$
  then let $\textit{match} = \iota(i \in \textsf{inds}\ \textit{cseq}) \cdot \textit{Match}(\textit{chooser}, \textit{cseq}[i].\textit{constset})$ in
    let $\mathbf{case}(\_, u) = \textit{cseq}[\textit{match}]$ in
     u
  else if $\textit{Has-Query}(\textit{chooser})$
   then $\mathbf{unitquery}(\textit{Type-Of-Unit}(\textit{unit}, \textit{sigdecs}))$
   else $\textit{unit}$

4.5 Evaluation of Primitive Functions

This section defines the semantics of the primitive function bodies. The definition of the delays and sample function have been taken from [HWM90a] with the definition of the Biops taken from [Tai88a].

To obtain the sequence of all relevant input values to any delay we need a function similar to *Update-Inputs*. This function, called *Get-Delay-Input*, differs from *Update-Inputs* since it does not necessarily have to go back to time zero. This is because a Delay has a finite history and hence there is a maximum depth of time that needs to be considered. The function *Get-Delay-Input* therefore collects together only those inputs to a Delay which will be needed for evaluation of a delay.

$$\textit{Get-Delay-Input}: \textit{kUnit} \times \textit{Signal}^{*} \times \textit{Input}^{*} \times \textit{kConst} \times \textit{Time} \times \textit{Time} \rightarrow \textit{kConst}^{*}$$

$\textit{Get-Delay-Input}(u, \textit{signals}, \textit{inputs}, \textit{initial}, t, s) \underline{\triangle}$
  if $t = s$
  then $[]$
  else if $t < 0$
   then $[\textit{initial}] \curvearrowright \textit{Get-Delay-Input}(u, \textit{signals}, \textit{inputs}, \textit{initial}, t\text{-}1, s)$
   else $[\textit{Evaluate-Unit}(u, \textit{signals}, \textit{inputs}, t)] \curvearrowright$
     $\textit{Get-Delay-Input}(u, \textit{signals}, \textit{inputs}, \textit{initial}, t\text{-}1, s)$

The function *Get-Delay-Input* is now incorporated into the following function which evaluates the result of the ambiguity delay primitives.

$$\textit{Evaluate-Delay}: \textit{Fnbody} \times \textit{Signaldec}^{*} \times \textit{Input}^{*} \times \textit{kUnit} \times \textit{Time} \rightarrow \textit{kConst}$$

$\textit{Evaluate-Delay}(\textit{delay}, \textit{sigdec}, \textit{inputs}, \textit{unit}, \textit{time}) \underline{\triangle}$
  let $\mathbf{delay}(\textit{initial}, m, \textit{ambig}, n) = \textit{delay}$ in
  let $r = [\textit{Min}(1, m), \cdots, m]$ in
  if $\textit{time} < 0$
  then $\textit{initial}$
  else let $\textit{dinput} = \textit{Get-Delay-Input}(\textit{unit}, \textit{signals}, \textit{inputs}, \textit{initial}, \textit{time}\text{-}1, \textit{time}\text{-}m\text{-}n)$ in
   if $\exists\, j \in r \cdot \forall\, i \in [1..n]\textit{dinput}[n + i\text{-}j] = \textit{dinput}[n]$
   then $\textit{dinput}[n]$
   else if $\exists\, j \in r \cdot \forall\, i \in [1..n]\textit{dinput}[n + i\text{-}j] = \textit{dinput}[n] \vee \mathbf{constquery}(\_)$
    then $\mathbf{constquery}(\textit{Type-Of-Const}(\textit{ambig}))$
    else $\textit{ambig}$

An inertial delay behaves in a different way to an ambiguity delay in that it filters out any input which is not stable for at least the length of the specified delay period (see [HWM90a] for a complete definition of the delay primitive). The resulting evaluation function is given by

$$\textit{Evaluate-Idelay}: \textit{Fnbody} \times \textit{Signaldec}^{*} \times \textit{Input}^{*} \times \textit{kUnit} \times \textit{Time} \rightarrow \textit{kConst}$$

$\textit{Evaluate-Idelay}(\textit{idelay}, \textit{sigdec}, \textit{inputs}, \textit{unit}, \textit{time}) \triangleq$
let $\mathbf{idelay}(\textit{initial}, n) = \textit{idelay}$ in
if $\textit{time} < 0$
then $\textit{initial}$
else let $\textit{dinput} = \textit{Get-Delay-Input}(\textit{unit}, \textit{signals}, \textit{inputs}, \textit{initial}, \textit{time}\text{-}1, \textit{time}\text{-}n)$ in
if $\forall\ i \in [1..n]\textit{dinput}[1] = \textit{dinput}[i]$
then $\textit{dinput}[n]$
else if $\forall\ i \in [1..n]\textit{dinput}[1] = \textit{dinput}[i] \vee \mathbf{constquery}(\_)$
then $\mathbf{constquery}(\textit{Type-of-Const}(\textit{initial}))$
else $\textit{Evaluate-Idelay}(\textit{idelay}, \textit{sigdec}, \textit{inputs}, \textit{unit}, \textit{time}\text{-}1)$

The REFORM construct is a way of taking a structured input and restructuring it. The function for performing this is defined below where *cseq* is the input structure of constant values, which have been flattened to its lowest level.

$$\textit{Evaluate-Reform}: \textit{kConst}^{*} \times \textit{kType} \rightarrow \textit{kConst}^{*} \times \textit{kConst}$$

$\textit{Evaluate-Reform}(\textit{cseq}, t) \triangleq$
if len $\textit{cseq} = 1 \wedge \textit{Has-Query}(\textit{cseq}[1])$
then $\mathbf{constquery}(t)$
else cases $\textit{Get-Type}(t)$ of
$\mathbf{types}([t_1, \cdots, t_k]) \rightarrow$ let $cs_{\cdot 1} = \textit{cseq}$ in
let $(cs_i, c_i) = \textit{Evaluate-Reform}(cs_{i\text{-}1}, t_i)$ $\quad i \in \{1..n\}$ in
$(cs_n,\ \mathbf{consts}([c_1, \cdots, c_n]))$
others (tl *cseq*, hd *cseq*)
end

The SAMPLE construct is a sample-and-hold primitive which samples its input at specified intervals and holds that value over the interval. The formal definition of this construct can be given by

$$\textit{Evaluate-Sample}: \textit{Fnbody} \times \textit{Signaldec}^{*} \times \textit{Input}^{*} \times \textit{kUnit} \times \textit{Time} \rightarrow \textit{kConst}$$

$\textit{Evaluate-Sample}(\textit{sample}, \textit{sigdec}, \textit{inputs}, \textit{unit}, \textit{time}) \triangleq$
let $\mathbf{sample}(\textit{interval}, \textit{initial}, \textit{skew}) = \textit{sample}$ in
if $\textit{time} < 0$
then $\textit{initial}$
else let $t' = t\text{-}((t\text{-}\textit{skew})\textit{MODinterval})$ in
$\textit{Evaluate-Unit}(\textit{unit}, \textit{sigdec}, \textit{inputs}, t')$

The RAM construct is a general read/write memory device. The size of the Ram is specified by

its enclosing function definition, elements of which can be written to at each simulation time. In order to define the evaluation of a Ram for a given address two auxiliary functions are needed. The first function, *Get-Ram-History*, behaves like *Update-Inputs* and calculates all the inputs to the Ram from time zero. The function is similar to *Update-Inputs* but uses concatenation since there is no local function calls within a RAM function

$\mathit{Get\text{-}Ram\text{-}History}\colon \mathit{kUnit} \times \mathit{Signaldec}^* \times \mathit{Input}^* \times \mathit{Time} \rightarrow \mathit{Input}^*$

$\mathit{Get\text{-}Ram\text{-}History}(\mathit{unit}, \mathit{sigdec}, \mathit{inputs}, \mathit{time}) \triangleq$
    if $\mathit{time} = 0$
    then $[]$
    else $[(\mathit{Evaluate\text{-}Unit}(\mathit{unit}, \mathit{sigdec}, \mathit{inputs}, \mathit{time}), \mathit{time})] \frown$
            $\mathit{Get\text{-}Ram\text{-}History}(\mathit{unit}, \mathit{sigdec}, \mathit{inputs}, \mathit{time}\text{-}1)$

The following function searches through a sequence of inputs to a ram for the desired read address. Since the list will be traversed starting at the most recent entry (in time) the first value located with the correct read address will be the last input to that address. Hence the explicit interrogation of time is not required.

$\mathit{Search\text{-}History}\colon \mathit{Input}^* \times \mathit{kConst} \times \mathit{kConst} \rightarrow \mathit{kConst}$

$\mathit{Search\text{-}History}(\mathit{history}, \mathit{initial}, \mathit{read}) \triangleq$
    if $\mathit{history} = []$
    then $\mathit{initial}$
    else if $(\mathsf{hd}\ \mathit{history})[1] = \mathbf{consts}([d, \mathit{read}, \_, \mathbf{enum}(\_, 1)])$
        then d
        else if $(\mathsf{hd}\ \mathit{history})[1] = \mathbf{consts}([\_, \mathbf{constquery}(\_), \_, \mathbf{constquery}(\_)]) \vee$
                        $\mathbf{consts}([\_, \mathbf{constquery}(\_), \_, \mathbf{enum}(\_, 1)]) \vee$
                        $\mathbf{consts}([\mathbf{constquery}(\_), \mathit{read}, \_, \mathbf{constquery}(\_)])$
            then $\mathbf{constquery}(\mathit{Type\text{-}Of\text{-}Const}(\mathit{initial}))$
            else $\mathit{Search\text{-}History}(\mathsf{tl}\ \mathit{history}, \mathit{initial}, \mathit{read})$

The above two functions can now be combined into the RAM evaluation function which is defined as

$\mathit{Evaluate\text{-}Ram}\colon \mathit{kConst} \times \mathit{Signaldec}^* \times \mathit{Input}^* \times \mathit{kUnit} \times \mathit{Time} \rightarrow \mathit{kConst}$

$\mathit{Evaluate\text{-}Ram}(\mathit{initial}, \mathit{sigdec}, \mathit{inputs}, \mathit{unit}, \mathit{time}) \triangleq$
    let $\mathbf{consts}([\mathit{data}, \mathit{write}, \mathit{read}, \mathit{enable}]) = \mathit{Evaluate\text{-}Unit}(\mathit{unit}, \mathit{sigdec}, \mathit{inputs}, \mathit{time})$ in
    if $\mathit{Get\text{-}Type}(\mathit{enable}) = \mathbf{enum}(\_, 1) \vee \mathit{Type\text{-}Equals}(\mathit{write}, \mathit{read})$
    then data
    else if $\mathit{Has\text{-}Query}(\mathit{read}) \vee \mathit{Has\text{-}Query}(\mathit{write}) \vee \mathit{Has\text{-}Query}(\mathit{enable})$
        then $\mathbf{constquery}(\mathit{Type\text{-}Of\text{-}Const}(\mathit{initial}))$
        else let $\mathit{ramhistory} = \mathit{Get\text{-}Ram\text{-}History}(\mathit{unit}, \mathit{sigdec}, \mathit{inputs}, \mathit{time})$ in
            $\mathit{Search\text{-}History}(\mathit{ramhistory}, \mathit{initial}, \mathit{read})$

Within the **Kernel** there are a number of Built in Operators (BIOP) which perform operations on enumerated types and bit strings. In the following section we define the semantics for each BIOP, here we merely state the enclosing calling function for a BIOP

Evaluate-Biop: *Fndec* × *kConst* → *kConst*

```
Evaluate-Biop(fdec, const) ≜
    if Has-Query(const) ∧ ¬ Optimal-Biop(fdec.fnbody.biopname)
    then constquery(Get-Type(fdec.outputtype))
    else let intype = Get-Type(fdec.inputtype) in
        let outtype = Get-Type(fdec.outputtype) in
        cases fdec.fnbody.biopname of
        AND           → Biop-And(const, intype, outtype)
        OR            → Biop-Or(const, intype, outtype)
        XOR           → Biop-Xor(const, intype, outtype)
        NOT           → Biop-Not(const, intype, outtype)
        EQ            → Biop-Eq(const, outtype)
        GT            → Biop-Gt(const, outtype)
        GE            → Biop-Ge(const, outtype)
        LT            → Biop-Lt(const, outtype)
        LE            → Biop-Le(const, outtype)
        EQ_US         → Biop-USeq(const, outtype)
        GT_US         → Biop-USgt(const, outtype)
        GE_US         → Biop-USge(const, outtype)
        LT_US         → Biop-USlt(const, outtype)
        LE_US         → Biop-USle(const, outtype)
        EQ_S          → Biop-Seq(const, outtype)
        GT_S          → Biop-Sgt(const, outtype)
        GE_S          → Biop-Sge(const, outtype)
        LT_S          → Biop-Slt(const, outtype)
        LE_S          → Biop-Sle(const, outtype)
        SL            → Biop-SL(const, outtype)
        SR_US         → Biop-SRus(const, outtype)
        SR_S          → Biop-SRs(const, outtype)
        PLUS_US       → Biop-USplus(const, outtype)
        MINUS_US      → Biop-USminus(const, outtype)
        NEGATE_US     → Biop-USneg(const, outtype)
        TIMES_US      → Biop-UStimes(const, outtype)
        DIVIDE_US     → Biop-USdivide(const, outtype)
        SQRT_US       → Biop-USsqrt(const, outtype)
        MOD_US        → Biop-USmod(const, outtype)
        RANGE_US      → Biop-USrange(const, outtype)
        PLUS_S        → Biop-Splus(const, outtype)
        MINUS_S       → Biop-Sminus(const, outtype)
        NEGATE_S      → Biop-Sneg(const, outtype)
        TIMES_S       → Biop-Stimes(const, outtype)
        DIVIDE_S      → Biop-Sdivide(const, outtype)
        ABS_S         → Biop-Sabs(const, outtype)
        MOD_S         → Biop-Smod(const, outtype)
        RANGE_S       → Biop-Srange(const, outtype)
        TRANSFORM_US  → Biop-TUS(const, intype, outtype)
        TRANSFORM_S   → Biop-TS(const, intype, outtype)

        end
```

4.6 Built-in Operators

In this section we present the semantics of all the Built-in Operators included within the **Kernel**. In particular these definitions express the way in which the type-ambiguity value is handled. For a fuller description of all ELLA's BIOPs see [Tai88b].

Throughout this section it is assumed that the input to a BIOP is a constant expression, called '*c*'. Where necessary the input and output type of a BIOP are passed into the semantic definitions and in those cases they are referred to as '*intype*' and '*outtype*', respectively.

4.6.1 Auxiliary Functions

This section defines some functions which are necessary for the definition of the BIOPs.

Enumerated type selection:

$\mathbf{e2b}\colon kConst \rightarrow \mathsf{B}$

$\mathbf{e2b}(c) \underline{\triangle}$
  if $\mathit{Check\text{-}Two\text{-}Val}(\mathit{Type\text{-}Of\text{-}Const}(c))$
  then $c = \mathbf{enum}(\_, 2)$
  else $\perp$

$\mathbf{b2e}_t\colon \mathsf{B} \rightarrow kConst$

$\mathbf{b2e}_t(b) \underline{\triangle}$
  if $\mathit{Check\text{-}Two\text{-}Val}(t)$
  then let $\mathbf{typeno}(typeno) = \mathit{Get\text{-}Type}(t)$ in
    $\mathbf{enum}(typeno, \text{if } b \text{ then } 2 \text{ else } 1)$
  else $\mathbf{constquery}(t)$

Bit-String conversion: (functions not shown)

$\mathit{Int\text{-}2\text{-}Sbit}(n, i)$	=	Integer i to signed bit string of length n
$\mathit{Int\text{-}2\text{-}Ubit}(n, i)$	=	Integer i to unsigned bit string of length n
$\mathit{Sbit\text{-}2\text{-}Int}(b)$	=	Signed bit string b to integer value
$\mathit{Ubit\text{-}2\text{-}Int}(b)$	=	Unsigned bit string b to integer value

Unsigned bit string evaluation:

$\mathbf{ust}\colon kConst \rightarrow \mathbb{N}$

$\mathbf{ust}(c) \underline{\triangle}$
  let $\mathbf{stringtype}(n, \_) = \mathit{Type\text{-}Of\text{-}Const}(c)$ in
  let $\mathbf{string}(\_, [tg_1, \cdots, tg_n]) = \mathit{Conv\text{-}String}(c)$ in
  let $bit_i = tg_i\text{-}1 \qquad \forall\, i \in \{1..n\}$ in
  $\mathit{Ubit\text{-}2\text{-}Int}(''bit_1 \cdots bit_n'')$

$\mathbf{ust}^{-1}{}_t : \mathbb{N} \rightarrow kConst$

$\mathbf{ust}^{-1}{}_t(v) \underline{\triangle}$
  let $\mathbf{stringtype}(n, ty) = t$ in
  let $\mathbf{typeno}(typeno) = Get\text{-}Type(ty)$ in
  if $v < 2^{n+1}$
  then let $''b_1 \cdots b_n'' = Int\text{-}2\text{-}Ubit(n, v)$ in
    let $tg_i = b_i + 1 \qquad \forall\, i \in \{1..n\}$ in
    $\mathbf{string}(typeno, [tg_1, \cdots, tg_n])$
  else $\mathbf{constquery}(t)$

Signed Bit string evaluation:

$\mathbf{sst}: kConst \rightarrow \mathbb{N}$

$\mathbf{sst}(c) \underline{\triangle}$
  let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c)$ in
  let $\mathbf{string}(\_, [tg_1, \cdots, tg_n]) = Conv\text{-}String(c)$ in
  let $bit_i = tg_i\text{-}1 \qquad \forall\, i \in \{1..n\}$ in
  $Sbit\text{-}2\text{-}Int(''bit_1 \cdots bit_n'')$

$\mathbf{sst}^{-1}{}_t : \mathbb{N} \rightarrow kConst$

$\mathbf{sst}^{-1}{}_t(v) \underline{\triangle}$
  let $\mathbf{stringtype}(n, ty) = t$ in
  let $\mathbf{typeno}(typeno) = Get\text{-}Type(ty)$ in
  if $-2^{n-1} \leq v < 2^{n-1}$
  then let $''b_1 \cdots b_n'' = Int\text{-}2\text{-}Sbit(n, v)$ in
    let $tg_i = b_i + 1 \qquad \forall\, i \in \{1..n\}$ in
    $\mathbf{string}(typeno, [tg_1, \cdots, tg_n])$
  else $\mathbf{constquery}(t)$

Quotient Evaluation: (quotient function not shown)

$OVER: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$OVER(n_1, n_2) \underline{\triangle}$
  if $n_2 \neq 0$
  then "Quotient of n_1 over n_2''
  else $\perp$

4.6.2 Biop Evaluation

4.6.2.1 Logical Operators

$Biop\text{-}And\colon kConst \times kType \times kType \rightarrow kConst$

$Biop\text{-}And(c, intype, outtype) \triangleq$
  if $Get\text{-}Type(intype) = \mathbf{types}([\_,\_])$
  then $Biop\text{-}EAnd(c, outtype)$
  else $Biop\text{-}SAnd(c, outtype)$

$Biop\text{-}Or\colon kConst \times kType \times kType \rightarrow kConst$

$Biop\text{-}Or(c, intype, outtype) \triangleq$
  if $Get\text{-}Type(intype) = \mathbf{types}([\_,\_])$
  then $Biop\text{-}EOr(c, outtype)$
  else $Biop\text{-}SOr(c, outtype)$

$Biop\text{-}Xor\colon kConst \times kType \times kType \rightarrow kConst$

$Biop\text{-}Xor(c, intype, outtype) \triangleq$
  if $Get\text{-}Type(intype) = \mathbf{types}([\_,\_])$
  then $Biop\text{-}EXor(c, outtype)$
  else $Biop\text{-}SXor(c, outtype)$

$Biop\text{-}Not\colon kConst \times kType \times kType \rightarrow kConst$

$Biop\text{-}Not(c, intype, outtype) \triangleq$
  if $Get\text{-}Type(intype) \neq \mathbf{stringtype}(\_,\_)$
  then $Biop\text{-}ENot(c, outtype)$
  else $Biop\text{-}SNot(c, outtype)$

Logical Operators on enumerated values:

$Biop\text{-}EAnd\colon kConst \times kType \rightarrow kConst$

$Biop\text{-}EAnd(c, outtype) \triangleq$
  let $\mathbf{consts}(c_1 c_2) = c$ in
  if $Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c_1)) \wedge Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c_2)) \wedge$
    $Check\text{-}Two\text{-}Val(outtype)$
  then $\mathbf{b2e}_{outtype}(\mathbf{e2b}(c_1) \wedge \mathbf{e2b}(c_2))$
  else $\mathbf{constquery}(outtype)$

$Biop\text{-}EOr\colon kConst \times kType \rightarrow kConst$

$Biop\text{-}EOr(c, outtype) \triangleq$
  let $\mathbf{consts}(c_1, c_2) = c$ in
  if $Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c_1)) \wedge Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c_2)) \wedge$
    $Check\text{-}Two\text{-}Val(outtype)$
  then $\mathbf{b2e}_{outtype}(\mathbf{e2b}(c_1) \vee \mathbf{e2b}(c_2))$
  else $\mathbf{constquery}(outtype)$

$Biop\text{-}EXor$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}EXor(c, outtype) \triangleq$
let $\mathbf{consts}(c_1, c_2) = c$ in
if $Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c_1)) \wedge Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c_2)) \wedge$
$Check\text{-}Two\text{-}Val(outtype)$
then $\mathbf{b2e}_{outtype}(\mathbf{e2b}(c_1) \oplus \mathbf{e2b}(c_2))$
else $\mathbf{constquery}(outtype)$

$Biop\text{-}ENot$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}ENot(c, outtype) \triangleq$
if $Check\text{-}Two\text{-}Val(Type\text{-}Of\text{-}Const(c)) \wedge Check\text{-}Two\text{-}Val(outtype)$
then $\mathbf{b2e}_{outtype}(\neg\ \mathbf{e2b}(c))$
else $\mathbf{constquery}(outtype)$

Logical Operators on unsigned strings:

$Biop\text{-}SAnd$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}SAnd(c, outtype) \triangleq$
let $\mathbf{consts}(c_1, c_2) = c$ in
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_1)$ in
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_2)$ in
let $\mathbf{stringtype}(n, ty) = outtype$ in
let $\mathbf{typeno}(typeno) = Get\text{-}Type(ty)$ in
$\mathbf{string}(typeno, ([\,\mathbf{b2e}_{ty}(\mathbf{e2b}(c_1[1]) \wedge \mathbf{e2b}(c_2[1])), \cdots, \mathbf{b2e}_{ty}(\mathbf{e2b}(c_1[n]) \wedge \mathbf{e2b}(c_2[n]))]))$

$Biop\text{-}SOr$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}SOr(c, outtype) \triangleq$
let $\mathbf{consts}(c_1, c_2) = c$ in
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_1)$ in
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_2)$ in
let $\mathbf{stringtype}(n, ty) = outtype$ in
let $\mathbf{typeno}(typeno) = Get\text{-}Type(ty)$ in
$\mathbf{string}(typeno, ([\,\mathbf{b2e}_{ty}(\mathbf{e2b}(c_1[1]) \vee \mathbf{e2b}(c_2[1])), \cdots, \mathbf{b2e}_{ty}(\mathbf{e2b}(c_1[n]) \vee \mathbf{e2b}(c_2[n]))]))$

$Biop\text{-}SXor$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}SXor(c, outtype) \triangleq$
let $\mathbf{consts}(c_1, c_2) = c$ in
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_1)$ in
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_2)$ in
let $\mathbf{stringtype}(n, ty) = outtype$ in
let $\mathbf{typeno}(typeno) = Get\text{-}Type(ty)$ in
$\mathbf{string}(typeno, ([\,\mathbf{b2e}_{ty}(\mathbf{e2b}(c_1[1]) \oplus \mathbf{e2b}(c_2[1])), \cdots, \mathbf{b2e}_{ty}(\mathbf{e2b}(c_1[n]) \oplus \mathbf{e2b}(c_2[n]))]))$

Biop-SNot: *kConst* × *kType* → *kConst*

Biop-SNot(*c*, *outtype*) $\triangleq$
let **stringtype**(n, _) = *Type-Of-Const*(c) in
let **stringtype**(n, ty) = *outtype* in
let **typeno**(*typeno*) = *Get-Type*(ty) in
string(*typeno*, ([$\mathbf{b2e}_{ty}(\neg\ \mathbf{e2b}(c[1]))$, ⋯, $\mathbf{b2e}_{ty}(\neg\ \mathbf{e2b}(c[n]))$]))

4.6.2.2 Relational Operators on Enumerated Types

Biop-Eq: *kConst* × *kType* → *kConst*

Biop-Eq(*c*, *outtype*) $\triangleq$
let **consts**(**enum**(_, tag_1), **enum**(_, tag_2)) = c in
if *Check-Two-Val*(*outtype*)
then $\mathbf{b2e}_t(tag_1 = tag_2)$
else **constquery**(*outtype*)

Biop-Gt: *kConst* × *kType* → *kConst*

Biop-Gt(*c*, *outtype*) $\triangleq$
let **consts**(**enum**(_, tag_1), **enum**(_, tag_2)) = c in
if *Check-Two-Val*(*outtype*)
then $\mathbf{b2e}_{outtype}(tag_1 > tag_2)$
else **constquery**(*outtype*)

Biop-Ge: *kConst* × *kType* → *kConst*

Biop-Ge(*c*, *outtype*) $\triangleq$
let **consts**(**enum**(_, tag_1), **enum**(_, tag_2)) = c in
if *Check-Two-Val*(*outtype*)
then $\mathbf{b2e}_{outtype}(tag_1 \geq tag_2)$
else **constquery**(*outtype*)

Biop-Lt: *kConst* × *kType* → *kConst*

Biop-Lt(*c*, *outtype*) $\triangleq$
let **consts**(**enum**(_, tag_1), **enum**(_, tag_2)) = c in
if *Check-Two-Val*(*outtype*)
then $\mathbf{b2e}_t(tag_1 < tag_2)$
else **constquery**(*outtype*)

$Biop\text{-}Le: kConst \times kType \rightarrow kConst$

$Biop\text{-}Le(c, outtype) \triangleq$
  let **consts**(**enum**(_, tag_1), **enum**(_, tag_2)) = c in
  if $Check\text{-}Two\text{-}Val(outtype)$
  then $\mathbf{b2e}_{outtype}(tag_1 \leq tag_2)$
  else **constquery**($outtype$)

4.6.2.3 Relational Operators on Unsigned Bit Strings

Relational optimisation test on unsigned bit strings:

$B\text{-}U\text{-}Test: \mathbb{N} \times \mathbb{N} \times kConst \rightarrow \mathbb{B}$

$B\text{-}U\text{-}Test(m, n, c) \triangleq$
  $(m > n) \wedge (\ \mathbf{string}(\_, [tg_1, \cdots, tg_m]) = Conv\text{-}String(c)) \wedge$
    $(\exists\ i \in \{1..(m\text{-}n)\} \cdot tg_i = 2)$

$Biop\text{-}USeq: kConst \times kType \rightarrow kConst$

$Biop\text{-}USeq(c, outtype) \triangleq$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = $Type\text{-}Of\text{-}Const(c_1)$ in
  let **stringtype**(m, _) = $Type\text{-}Of\text{-}Const(c_2)$ in
  if $B\text{-}U\text{-}Test(m, n, c_2) \vee B\text{-}U\text{-}Test(n, m, c_1)$
  then $\mathbf{b2e}_{outtype}(\mathsf{false})$
  else if $Has\text{-}Query(c_1) \vee Has\text{-}Query(c_2)$
    then **constquery**($outtype$)
    else $\mathbf{b2e}_{outtype}(\ \mathbf{ust}(c_1) = \mathbf{ust}(c_2))$

$Biop\text{-}USgt: kConst \times kType \rightarrow kConst$

$Biop\text{-}USgt(c, outtype) \triangleq$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = $Type\text{-}Of\text{-}Const(c_1)$ in
  let **stringtype**(m, _) = $Type\text{-}Of\text{-}Const(c_2)$ in
  if $B\text{-}U\text{-}Test(m, n, c_2)$
  then $\mathbf{b2e}_{outtype}(\mathsf{false})$
  else if $B\text{-}U\text{-}Test(n, m, c_1)$
    then $\mathbf{b2e}_{outtype}(\mathsf{true})$
    else if $Has\text{-}Query(c_1) \vee Has\text{-}Query(c_2)$
      then **constquery**($outtype$)
      else $\mathbf{b2e}_{outtype}(\ \mathbf{ust}(c_1) > \mathbf{ust}(c_2))$

Biop-USge: *kConst* $\times$ *kType* $\rightarrow$ *kConst*

Biop-USge(*c*, *outtype*) $\triangleq$

let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = *Type-Of-Const*(c_1) in
let **stringtype**(m, _) = *Type-Of-Const*(c_2) in
if *B-U-Test*(m, n, c_2)
then $\mathbf{b2e}_{outtype}$(false)
else if *B-U-Test*(n, m, c_1)
  then $\mathbf{b2e}_{outtype}$(true)
  else if *Has-Query*(c_1) $\vee$ *Has-Query*(c_2)
    then **constquery**(*outtype*)
    else $\mathbf{b2e}_{outtype}$(**ust**(c_1)$\geq$ **ust**(c_2))

Biop-USlt: *kConst* $\times$ *kType* $\rightarrow$ *kConst*

Biop-USlt(*c*, *outtype*) $\triangleq$

let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = *Type-Of-Const*(c_1) in
let **stringtype**(m, _) = *Type-Of-Const*(c_2) in
if *B-U-Test*(n, m, c_1)
then $\mathbf{b2e}_{outtype}$(false)
else if *B-U-Test*(m, n, c_2)
  then $\mathbf{b2e}_{outtype}$(true)
  else if *Has-Query*(c_1) $\vee$ *Has-Query*(c_2)
    then **constquery**(*outtype*)
    else $\mathbf{b2e}_{outtype}$(**ust**(c_1) $<$ **ust**(c_2))

Biop-USle: *kConst* $\times$ *kType* $\rightarrow$ *kConst*

Biop-USle(*c*, *outtype*) $\triangleq$

let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = *Type-Of-Const*(c_1) in
let **stringtype**(m, _) = *Type-Of-Const*(c_2) in
if *B-U-Test*(n, m, c_1)
then $\mathbf{b2e}_{outtype}$(false)
else if *B-U-Test*(m, n, c_2)
  then $\mathbf{b2e}_{outtype}$(true)
  else if *Has-Query*(c_1) $\vee$ *Has-Query*(c_2)
    then **constquery**(*outtype*)
    else $\mathbf{b2e}_{outtype}$(**ust**(c_1)$\leq$ **ust**(c_2))

4.6.2.4 Relational Operators on Signed Bit Strings

Relational optimisation test on signed bit strings:

$B\text{-}S\text{-}Test$: $\mathbb{N} \times \mathbb{N} \times kConst \rightarrow \mathbb{B}$

$B\text{-}S\text{-}Test(m, n, c) \triangleq$
  $(m > n) \wedge (\textbf{string}(\_, [tg_1, \cdots, tg_m]) = Conv\text{-}String(c)) \wedge$
   $(tg_1 = 1) \wedge (\exists\, i \in \{2..(m\text{-}n)\} \cdot tg_i = 2)$

$Biop\text{-}Seq$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}Seq(c, outtype) \triangleq$
  let $\textbf{consts}(c_1, c_2) = c$ in
  let $\textbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_1)$ in
  let $\textbf{stringtype}(m, \_) = Type\text{-}Of\text{-}Const(c_2)$ in
  if $B\text{-}S\text{-}Test(m, n, c_2) \vee B\text{-}S\text{-}Test(n, m, c_1)$
  then $\textbf{b2e}_{outtype}(\text{false})$
  else if $Has\text{-}Query(c_1) \vee Has\text{-}Query(c_2)$
   then $\textbf{constquery}(outtype)$
   else $\textbf{b2e}_{outtype}(\textbf{sst}(c_1) = \textbf{sst}(c_2))$

$Biop\text{-}Sgt$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}Sgt(c, outtype) \triangleq$
  let $\textbf{consts}(c_1, c_2) = c$ in
  let $\textbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_1)$ in
  let $\textbf{stringtype}(m, \_) = Type\text{-}Of\text{-}Const(c_2)$ in
  if $B\text{-}S\text{-}Test(m, n, c_2)$
  then $\textbf{b2e}_{outtype}(\text{false})$
  else if $B\text{-}S\text{-}Test(n, m, c_1)$
   then $\textbf{b2e}_{outtype}(\text{true})$
   else if $Has\text{-}Query(c_1) \vee Has\text{-}Query(c_2)$
    then $\textbf{constquery}(outtype)$
    else $\textbf{b2e}_{outtype}(\textbf{sst}(c_1) > \textbf{sst}(c_2))$

$Biop\text{-}Sge$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}Sge(c, outtype) \triangleq$
  let $\textbf{consts}(c_1, c_2) = c$ in
  let $\textbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c_1)$ in
  let $\textbf{stringtype}(m, \_) = Type\text{-}Of\text{-}Const(c_2)$ in
  if $B\text{-}S\text{-}Test(m, n, c_2)$
  then $\textbf{b2e}_{outtype}(\text{false})$
  else if $B\text{-}S\text{-}Test(n, m, c_1)$
   then $\textbf{b2e}_{outtype}(\text{true})$
   else if $Has\text{-}Query(c_1) \vee Has\text{-}Query(c_2)$
    then $\textbf{constquery}(outtype)$
    else $\textbf{b2e}_{outtype}(\textbf{sst}(c_1) \geq \textbf{sst}(c_2))$

Biop-Slt: *kConst* × *kType* → *kConst*

Biop-Slt(c, *outtype*) $\underline{\triangle}$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = *Type-Of-Const*(c_1) in
  let **stringtype**(m, _) = *Type-Of-Const*(c_2) in
  if *B-S-Test*(n, m, c_1)
  then $\mathbf{b2e}_{outtype}$(false)
  else if *B-S-Test*(m, n, c_2)
    then $\mathbf{b2e}_{outtype}$(true)
    else if *Has-Query*(c_1) $\vee$ *Has-Query*(c_2)
      then **constquery**(*outtype*)
      else $\mathbf{b2e}_{outtype}$(**sst**(c_1) < **sst**(c_2))

Biop-Sle: *kConst* × *kType* → *kConst*

Biop-Sle(c, *outtype*) $\underline{\triangle}$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = *Type-Of-Const*(c_1) in
  let **stringtype**(m, _) = *Type-Of-Const*(c_2) in
  if *B-S-Test*(n, m, c_1)
  then $\mathbf{b2e}_{outtype}$(false)
  else if *B-S-Test*(m, n, c_2)
    then $\mathbf{b2e}_{outtype}$(true)
    else if *Has-Query*(c_1) $\vee$ *Has-Query*(c_2)
      then **constquery**(*outtype*)
      else $\mathbf{b2e}_{outtype}$(**sst**(c_1)$\leq$ **sst**(c_2))

4.6.2.5 Shift Operators on Unsigned and Signed Bit Strings

Biop-SL: *kConst* × *ktype* → *kConst*

Biop-SL(c, *outtype*) $\underline{\triangle}$
  let **stringtype**(n, ty) = *Type-Of-Const*(c) in
  let **stringtype**($n + m, ty$) = *outtype* in
  let **string**(*typeno*, $[tg_1, \cdots tg_n]$) = *Conv-String*(c) in
  string(*typeno*, $[tg_1, \cdots, tg_n] \frown [(1)^m]$)

Biop-SRus: *kConst* × *ktype* → *kConst*

Biop-SRus(c, *outtype*) $\underline{\triangle}$
  let **stringtype**(n, ty) = *Type-Of-Const*(c) in
  let **stringtype**($n + m, ty$) = *outtype* in
  let **string**(*typeno*, $[tg_1, \cdots tg_n]$) = *Conv-String*(c) in
  string(*typeno*, $[(1)^m] \frown [tg_1, \cdots, tg_n]$)

Biop-SRs: *kConst* × *ktype* → *kConst*

Biop-SRs(c, *outtype*) $\triangleq$
  let **stringtype**(n, ty) = *Type-Of-Const*(c) in
  let **stringtype**($n + m, ty$) = *outtype* in
  let **string**(*typeno*, $[tg_1, \cdots tg_n]$) = *Conv-String*(c) in
  string(*typeno*, $[tg_1{}^m] \frown [tg_1, \cdots, tg_n]$)

4.6.2.6 Arithmetic Operators on Unsigned Bit Strings

Biop-USplus: *kConst* × *kType* → *kConst*

Biop-USplus(c, *outtype*) $\triangleq$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = c_1 in
  let **stringtype**(m, _) = c_2 in
  let **stringtype**($MAX(m, n) + 1$, _) = *outtype* in
  $\mathbf{ust}^{-1}{}_{outtype}(\ \mathbf{ust}\,c_1 +\ \mathbf{ust}\,c_2)$

Biop-USminus: *kConst* × *kType* → *kConst*

Biop-USminus(c, *outtype*) $\triangleq$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = c_1 in
  let **stringtype**(m, _) = c_2 in
  let **stringtype**($MAX(m, n) + 1$, _) = *outtype* in
  $\mathbf{ust}^{-1}{}_{outtype}(\ \mathbf{ust}\,c_1 -\ \mathbf{ust}\,c_2)$

Biop-USneg: *kConst* × *kType* → *kConst*

Biop-USneg(c, *outtype*) $\triangleq$
  let **stringtype**(n, _) = c in
  let **stringtype**($n + 1$, _) = *outtype* in
  $\mathbf{ust}^{-1}{}_{outtype}(\neg\ \mathbf{ust}\,c)$

Biop-UStimes: *kConst* × *kType* → *kConst*

Biop-UStimes(c, *outtype*) $\triangleq$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = c_1 in
  let **stringtype**(m, _) = c_2 in
  let **stringtype**($m + n$, _) = *outtype* in
  $\mathbf{ust}^{-1}{}_{outtype}(\ \mathbf{ust}\,c_1 *\ \mathbf{ust}\,c_2)$

Biop-USdivide: *kConst* × *kType* → *kConst*

$\textit{Biop-USdivide}(c, \textit{outtype}) \triangleq$
$\quad$ let $\textbf{consts}(c_1, c_2) = c$ in
$\quad$ let $\textbf{stringtype}(n, \_) = c_1$ in
$\quad$ let $\textbf{stringtype}(m, \_) = c_2$ in
$\quad$ let $\textbf{types}(ty, ty1, ty2) = \textit{outtype}$ in
$\quad$ let $\textbf{stringtype}(n, \_) = ty1$ in
$\quad$ let $\textbf{stringtype}(m, \_) = ty2$ in
$\quad$ if $\textbf{ust}\, c_2 \neq 0$
$\quad$ then $\textbf{consts}([\ \textbf{b2e}_{ty}(\text{false}),\ \textbf{ust}^{-1}{}_{ty1}(\ \textbf{ust}\, c_1\, OVER\ \textbf{ust}\, c_2),$
$\quad\quad \textbf{ust}^{-1}{}_{ty2}((\ \textbf{ust}\, c_1\, OVER\ \textbf{ust}\, c_2) * \ \textbf{ust}\, c_2))])$
$\quad$ else $\textbf{consts}([\ \textbf{b2e}_{ty}(\text{true}),\ \textbf{constquery}(ty1),\ \textbf{constquery}(ty2)])$

Biop-USsqrt: *kConst* × *kType* → *kConst*

$\textit{Biop-USsqrt}(c, \textit{outtype}) \triangleq$
$\quad$ let $\textbf{stringtype}(n, \_) = c$ in
$\quad$ let $\textbf{stringtype}((n+1)\%2, \_) = \textit{outtype}$ in
$\quad \textbf{ust}^{-1}{}_{outtype}(\ \surd\ \textbf{ust}\, c)$

Biop-USmod: *kConst* × *kType* → *kConst*

$\textit{Biop-USmod}(c, \textit{outtype}) \triangleq$
$\quad$ let $\textbf{consts}(c_1, c_2) = c$ in
$\quad$ let $\textbf{stringtype}(n, \_) = c_1$ in
$\quad$ let $\textbf{stringtype}(m, \_) = c_2$ in
$\quad$ let $\textbf{types}([ty,\ \textbf{stringtype}(m, \_)]) = \textit{outtype}$ in
$\quad$ if $\textbf{ust}\, c_2 \neq 0$
$\quad$ then $\textbf{consts}([\ \textbf{b2e}_{ty}(\text{false}),\ \textbf{ust}^{-1}{}_{outtype[2]}(\ \textbf{ust}\, c_1\, MOD\ \textbf{ust}\, c_2)])$
$\quad$ else $\textbf{consts}([\ \textbf{b2e}_{ty}(\text{true}),\ \textbf{constquery}(\textit{outtype}[2]))$

Biop-USrange: *kConst* × *kType* → *kConst*

$\textit{Biop-USrange}(c, \textit{outtype}) \triangleq$
$\quad$ let $\textbf{stringtype}(n, \_) = c$ in
$\quad$ let $\textbf{types}(ty,\ \textbf{stringtype}(m, \_)) = \textit{outtype}$ in
$\quad$ if $\textbf{ust}\, c < 2^m$
$\quad$ then $\textbf{consts}([\ \textbf{b2e}_{ty}(\text{false}),\ \textbf{ust}^{-1}{}_{outtype[2]}(\ \textbf{ust}\, c)])$
$\quad$ else $\textbf{consts}([\ \textbf{b2e}_{ty}(\text{true}),\ \textbf{constquery}(\textit{outtype}[2])])$

4.6.2.7 Arithmetic Operators on Signed Bit Strings

Biop-Splus: *kConst* × *kType* → *kConst*

Biop-Splus(c, *outtype*) $\triangleq$
let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = c_1 in
let **stringtype**(m, _) = c_2 in
let **stringtype**($MAX(m, n) + 1$, _) = *outtype* in
$\mathbf{sst}^{-1}_{outtype}$($\mathbf{sst}\,c_1$ + $\mathbf{sst}\,c_2$)

Biop-Sminus: *kConst* × *kType* → *kConst*

Biop-Sminus(c, *outtype*) $\triangleq$
let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = c_1 in
let **stringtype**(m, _) = c_2 in
let **stringtype**($MAX(m, n) + 1$, _) = *outtype* in
$\mathbf{sst}^{-1}_{outtype}$($\mathbf{sst}\,c_1$- $\mathbf{sst}\,c_2$)

Biop-USneg: *kConst* × *kType* → *kConst*

Biop-USneg(c, *outtype*) $\triangleq$
let **stringtype**(n, _) = c in
let **stringtype**($n + 1$, _) = *outtype* in
$\mathbf{sst}^{-1}_{outtype}$($\neg$ $\mathbf{sst}\,c$)

Biop-Stimes: *kConst* × *kType* → *kConst*

Biop-Stimes(c, *outtype*) $\triangleq$
let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = c_1 in
let **stringtype**(m, _) = c_2 in
let **stringtype**($m + n$, _) = *outtype* in
$\mathbf{sst}^{-1}_{outtype}$($\mathbf{sst}\,c_1$ * $\mathbf{sst}\,c_2$)

Biop-Sdivide: *kConst* × *kType* → *kConst*

Biop-Sdivide(c, *outtype*) $\triangleq$
let **consts**(c_1, c_2) = c in
let **stringtype**(n, _) = c_1 in
let **stringtype**(m, _) = c_2 in
let **types**($ty, ty1, ty2$) = *outtype* in
let **stringtype**(n, _) = $ty1$ in
let **stringtype**(m, _) = $ty2$ in
if $\mathbf{sst}\,c_2 \neq 0$
then **consts**([$\mathbf{b2e}_{ty}$(false), $\mathbf{sst}^{-1}_{ty1}$($\mathbf{sst}\,c_1$ *OVER* $\mathbf{sst}\,c_2$),
$\mathbf{sst}^{-1}_{ty2}$(($\mathbf{sst}\,c_1$ *OVER* $\mathbf{sst}\,c_2$) * $\mathbf{sst}\,c_2$))])
else **consts**([$\mathbf{b2e}_{ty}$(true), **constquery**($ty1$), **constquery**($ty2$)])

Biop-Smod: *kConst* × *kType* → *kConst*

Biop-Smod(*c*, *outtype*) $\triangleq$
  let **consts**(c_1, c_2) = c in
  let **stringtype**(n, _) = c_1 in
  let **stringtype**(m, _) = c_2 in
  let **types**([ty, **stringtype**(m, _)]) = t in
  if **sst**c_2 = 0
  then **consts**([**b2e**$_{ty}$(true), **constquery**(*outtype*[2])])
  else **consts**([**b2e**$_{ty}$(false), **sst**$^{-1}_{outtype[2]}$(**sst**c_1 *MOD* **sst**c_2)])

Biop-Srange: *kConst* × *kType* → *kConst*

Biop-Srange(*c*, *outtype*) $\triangleq$
  let **stringtype**(n, _) = c in
  let **types**(ty, **stringtype**(m, _)) = *outtype* in
  if **sst**c < 2^m
  then **consts**([**b2e**$_{ty}$(false), **sst**$^{-1}_{outtype[2]}$(**sst**c)])
  else **consts**([**b2e**$_{ty}$(true), **constquery**(*outtype*[2])])

Biop-Sabs: *kConst* × *kType* → *kConst*

Biop-Sabs(*c*, *outtype*) $\triangleq$
  let **stringtype**(n, _) = c in
  let **stringtype**(n, _) = *outtype* in
  sst$^{-1}_{outtype}$(*ABS* **sst**c)

4.6.2.8 Transformation Operators

This section defines transformational operators on unsigned, signed strings and integer types

Biop-TUS: *kConst* × *kType* × *kType* → *kConst*

Biop-TUS(*c*, *intype*, *outtype*) $\triangleq$
  if *Get-Type*(*intype*) = **stringtype**(_, _)
  then *Biop-STUS*(*c*, *outtype*)
  else *Biop-ETUS*(*c*, *outtype*)

Biop-TS: *kConst* × *kType* × *kType* → *kConst*

Biop-TS(*c*, *intype*, *outtype*) $\triangleq$
  if *Get-Type*(*intype*) = **stringtype**(_, _)
  then *Biop-STS*(*c*, *outtype*)
  else *Biop-ETS*(*c*, *outtype*)

$Biop\text{-}STUS$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}STUS(c, outtype) \triangleq$
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c)$ in
let $\mathbf{types}(ty1, ty2) = outtype$ in
let $(\_, lwb, upb) = Find\text{-}Integer\text{-}Type(ty2)$ in
if $\mathbf{ust}\, c < upb$
then $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{false}),\ \mathbf{ust}\, c])$
else $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{true}),\ \mathbf{constquery}(ty2)])$

$Biop\text{-}ETUS$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}ETUS(c, outtype) \triangleq$
let $ty = Type\text{-}Of\text{-}Const(c)$ in
let $(\_, lwb, upb) = Find\text{-}Integer\text{-}Type(ty)$ in
let $\mathbf{enum}(\_, tagno) = c$ in
let $\mathbf{types}(ty1,\ \mathbf{stringtype}(m, \_)) = outtype$ in
let $int = lwb + tagno\text{-}1$ in
if $int < 2^m$
then $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{false}),\ \mathbf{ust}^{-1}{}_{outtype[2]}\, c])$
else $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{true}),\ \mathbf{constquery}(outtype[2])])$

$Biop\text{-}STS$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}STS(c, outtype) \triangleq$
let $\mathbf{stringtype}(n, \_) = Type\text{-}Of\text{-}Const(c)$ in
let $\mathbf{types}(ty1, ty2) = outtype$ in
let $(\_, lwb, upb) = Find\text{-}Integer\text{-}Type(ty2)$ in
if $\mathbf{sst}\, c < upb$
then $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{false}),\ \mathbf{sst}\, c])$
else $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{true}),\ \mathbf{constquery}(ty2)])$

$Biop\text{-}ETS$: $kConst \times kType \rightarrow kConst$

$Biop\text{-}ETS(c, outtype) \triangleq$
let $ty = Type\text{-}Of\text{-}Const(c)$ in
let $(\_, lwb, upb) = Find\text{-}Integer\text{-}Type(ty)$ in
let $\mathbf{enum}(\_, tagno) = c$ in
let $\mathbf{types}(ty1,\ \mathbf{stringtype}(m, \_)) = outtype$ in
let $int = lwb + tagno\text{-}1$ in
if $-2^{m-1} \leq int < 2^{m-1}$
then $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{false}),\ \mathbf{sst}^{-1}{}_{outtype[2]}\, c])$
else $\mathbf{consts}([\ \mathbf{b2e}_{ty1}(\text{true}),\ \mathbf{constquery}(outtype[2])])$

4.7 Function Evaluation

In the previous sections we have defined all the functions necessary for the evaluation of a **Kernel** function instance. It now remains to construct the appropriate calling functions. The outer most function will be one which takes a function instance with a specified input value list and a specified evaluation time and returns a constant value result. A function instance body can either be a basic built in function, such as Delay, Ram etc, or it can be a unit clause. The following two functions define the evaluation calling process for these two cases.

The evaluation of a unit expression is defined by

$$\textit{Evaluate-Unit}: \textit{Unit} \times \textit{Signaldec}^* \times \textit{Input}^* \times \textit{Time} \rightarrow \textit{Const}$$

$\textit{Evaluate-Unit}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time}) \triangleq$

cases *unit* of

conc(u_1, u_2, ty) → let $c_1 = \textit{Evaluate-Unit}(u_1, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
let $c_2 = \textit{Evaluate-Unit}(u_2, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
if **stringtype**$(\_,\_) = \textit{Get-Type}(ty)$
then $\textit{Conc-String}(c_1, c_2, ty)$
else $\textit{Conc-Const}(c_1, c_2, ty)$

unitstring$(\textit{size}, \textit{unit})$ → let $c = \textit{Evaluate-Unit}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
conststring$(\textit{size}, c)$

units$([u_1, \cdots, u_k])$ → let $c_i = \textit{Evaluate-Unit}(u_i, \textit{sigdecs}, \textit{inputs}, \textit{time})\ \forall\ i \in [1..k]$ in
consts$([c_1, \cdots, c_k])$

instance$(\textit{fnno}, \textit{unit})$ → let $\textit{fdec} = (\textit{EnvFndec})[\textit{fnno}]$ in
if $\textit{fdec.fnbody} \in k\textit{Unit}$
then let $\textit{inputs}' = \textit{Update-Inputs}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
$\textit{Evaluate-Fn}(\textit{fnno}, \textit{inputs}', \textit{time})$
else $\textit{Evaluate-Builtin}(\textit{fdec}, \textit{sigdecs}, \textit{inputs}, \textit{unit}, \textit{time})$

unitassoc$(\textit{enum}, \textit{unit})$ → let $c = \textit{Evaluate-Unit}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
constassoc$(\textit{enum}, c)$

extract$(\textit{unit}, \textit{enum})$ → let $c = \textit{Evaluate-Unit}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
$\textit{Evaluate-Extract}(c, \textit{enum})$

signal$(\textit{signalno})$ → $\textit{Sig}(\textit{signalno}, \textit{sigdecs}, \textit{inputs}, \textit{time})$

index$(\textit{unit}, \textit{ind}, ty)$ → let $c = \textit{Evaluate-Unit}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
$\textit{Evaluate-Index}(c, \textit{ind}, ty)$

trim$(\textit{unit}, \textit{ind}1, \textit{ind}2, ty)$ → let $c = \textit{Evaluate-Unit}(\textit{unit}, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
$\textit{Evaluate-Trim}(c, \textit{ind}1, \textit{ind}2, ty)$

dyindex(u_1, u_2, ty) → let $c_1 = \textit{Evaluate-Unit}(u_1, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
let $c_2 = \textit{Evaluate-Unit}(u_2, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
$\textit{Evaluate-Dyindex}(c_1, c_2, ty)$

replace(u_1, u_2, u_3) → let $c_1 = \textit{Evaluate-Unit}(u_1, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
let $c_2 = \textit{Evaluate-Unit}(u_2, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
let $c_3 = \textit{Evaluate-Unit}(u_3, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
$\textit{Evaluate-Replace}(c_1, c_2, c_3)$

unitquery(ty) → **constquery**(ty)

caseclause$(u_1, \textit{cseq}, u_2)$ → let $\textit{chooser} = \textit{Evaluate-Unit}(u_1, \textit{sigdecs}, \textit{inputs}, \textit{time})$ in
let $u = \textit{Evaluate-Case}(\textit{chooser}, \textit{cseq}, u_2, \textit{sigdecs})$ in
$\textit{Evaluate-Unit}(u, \textit{sigdecs}, \textit{inputs}, \textit{time})$

unitvoid → **constvoid**

end

and the evaluation of the built-in primitives is defined by

$$\textit{Evaluate-Builtin}: kFndec \times Signaldec^* \times Input^* \times kUnit \times Time \rightarrow kConst$$

$\textit{Evaluate-Builtin}(fdec, sigdec, inputs, unit, time) \triangleq$

cases $fdec.fnbody$ of

reform $\rightarrow$ let $cseq = \textit{Flatten-Const}(\textit{Evaluate-Unit}(unit, sigdec, inputs, time))$ in
$\quad\textit{Evaluate-Reform}(cseq, fdec.outputtype)[2]$

biop $\rightarrow \textit{Evaluate-Biop}(fdec, \textit{Evaluate-Unit}(unit, sigdec, inputs, time))$

delay $\rightarrow \textit{Evaluate-Delay}(fdec.fnbody, sigdec, inputs, unit, time)$

idelay $\rightarrow \textit{Evaluate-Idelay}(fdec.fnbody, sigdec, inputs, unit, time)$

sample $\rightarrow \textit{Evaluate-Sample}(fdec.fnbody, sigdec, inputs, unit, time)$

ram $\rightarrow \textit{Evaluate-Ram}(fdec.fnbody.initial, sigdec, inputs, unit, time)$

end

For completeness we restate the function which calculates all the necessary inputs for a function call, this function is the same as defined for K2.

$$\textit{Update-Inputs}: Expr \times Signal^* \times Input^* \times Time \rightarrow Input^*$$

$\textit{Update-Inputs}(e, signals, inputs, t) \triangleq$

if $t \geq 0$

then let $inputs' = \textit{Update-Inputs}(e, signals, inputs, (t\text{-}1))$ in
$\quad inputs' \dagger [(\textit{Evaluate-Exp}(e, signals, inputs', t), t)]$

else $inputs$

We can now define the top level function which evaluates a **Kernel** function instance at a given time and with a given input history

$$\textit{Evaluate-Fn}: Nat \times Input^* \times Time \rightarrow Const$$

$\textit{Evaluate-Fn}(fnno, inputs, time) \triangleq$

let $fdec = (EnvFndec)[fnno]$ in

if $fdec.fnbody \in kUnit$

then $\textit{Evaluate-Unit}(fdec.fnbody, fdec.signaldecseq, inputs, time)$

else let $(c, time) \in inputs$ in
$\quad\textit{Evaluate-Builtin}(fdec, fdec.signaldecseq, inputs, \textit{Convert-Const-Unit}(c), time)$

This now completes the definition of the dynamic semantics of the **Kernel**. In the next subsection we revisit an example shown previously to show how an ELLA description transforms into a **Kernel** expression and what the input to its evaluation function would be.

4.8 Example

In this example we present the example of a Reset/Set Flip Flop as given in section 3.4. An ELLA description of this circuit is

```
TYPE bool = NEW (h | l).

FN DEL = (bool)->bool: DELAY(l,1,l,1).

FN NOR = ([2]bool:in) -> bool:
CASE in OF
  (l,l) : h,
  (l,h) : l,
  (h,l) : l,
  (h,h) : l
  ELSE l
ESAC.

FN RSFF = (bool: in1, bool: in2) -> bool:
BEGIN
   MAKE DEL: del1.
   MAKE DEL: del2.
   MAKE NOR: nor1.
   MAKE NOR: nor2.
   JOIN (in1, del2) -> nor1.
   JOIN (in2, del1) -> nor2.
   JOIN nor1 -> del1.
   JOIN nor2 -> del2.
  OUTPUT del1
END.
```

The resulting **Kernel** environment, which is obtained by passing the above ELLA description of the circuit through the implementation of the semantic rules defined in [HM92], is given by

```
([TYPEDEC ("bool"  Tags([Tag(h, NIL),Tag(l, NIL)]))],
 [FNDEC( DEL, Typeno(1), [], Typeno(1), Delay(Enum(1, 2), 1, Enum(1, 2), 1))
  FNDEC( NOR, Types([Typeno(1), Typeno(1)]),
             [Signaldec("in", Types([Typeno(1), Typeno(1)]), input)],
             Typeno(1),
             Caseclause(Signal(1),
                        [Case(Constsets([Enum(1, 2),Enum(1, 2)]), Enum(1, 1)),
                         Case(Constsets([Enum(1, 2),Enum(1, 1)]), Enum(1, 2)),
                         Case(Constsets([Enum(1, 1),Enum(1, 2)]), Enum(1, 2)),
                         Case(Constsets([Enum(1, 1),Enum(1, 1)]), Enum(1, 2))],
                        Enum(1, 2)))
  FNDEC( RSFF, Types([Typeno(1), Typeno(1)]),
              [Signaldec("in1", Typeno(1), input),
               Signaldec("in2", Typeno(1), input),
               Signaldec("del1", Typeno(1), Instance(1, Signal(5))),
               Signaldec("del2", Typeno(1), Instance(1, Signal(6))),
               Signaldec("nor1", Typeno(1), Instance(2, Units([Signal(1), Signal(4)]))),
               Signaldec("nor2", Typeno(1), Instance(2, Units([Signal(2),Signal(3)])))],
              Typeno(1),
              Signal(3))
 ])
```

The correspondence with the environment given for this circuit in section 3.4, for the language K2, can be noted. Evaluation of this circuit follows the approach given for K2 e.g. Evaluation of RSFF at time t=3 is

$$Evaluate\text{-}Fn(3, inputs, 3)$$

where

```
inputs  =  [  ( consts( enum(1,2),  enum(1,1)), 0),   # (l,h) #
              ( consts( enum(1,2),  enum(1,1)), 1),   # (l,h) #
              ( consts( enum(1,2),  enum(1,1)), 2),   # (l,h) #
              ( consts( enum(1,2),  enum(1,1)), 3)    # (l,h) #
           ]
```

The result of the evaluation would be **enum(1,1)** # h #.

5 Conclusion

In this report we have defined the dynamic semantics of Kernel ELLA. A link between the work described here and earlier investigations has been shown. An implementation of the rules for a reduced Kernel language, called K2, has been carried out. The semantics given for the **Kernel** in this document are meant to reflect the semantics of the full ELLA system, however as only limited checking of the definitions given here has been carried out their correctness or otherwise remains to be established.

6 Acknowledgements

The author would like to acknowledge the support of the IED project 4/1/1357 "Formal Verification Support for ELLA".

A Glossary of Symbols

Functions

$f: D_1 \times D_2 \rightarrow R$	signature
$f \triangleq \cdots$	function definition
$f(\overline{d})$	application
if $\cdots$ *then* $\cdots$ *else* $\cdots$	condtional
let $x = \cdots$ *in* $\cdots$	local definition
case x *of* $\cdots$ *else* $\cdots$ *end*	choice
post	post-condition
ext rd	external read

Sets

T-set	finite subset of T
$\{t_1, \cdots, t_k\}$	set enumeration
$\{\}$	empty set
$t \in T$	set membership
$T_1 \cap T_2$	set intersection
$T_1 \cup T_2$	set union
$T_1 \subseteq T_2$	set containment
$T_1 \dagger T_2$	overwriting
$\mathbb{Z}$	$\{\cdots, -1, 0, 1, \cdots\}$
$\mathbb{N}_1$	$\{1, 2, \cdots\}$
$\mathbb{B}$	{true, false}

Sequences

S^*	finite sequence
$[s_1, \cdots, s_k]$	sequence enumeration
$[\,]$	empty sequence
len l	length of sequence l
$s_1 \frown s_2$	concaternation
$\iota\,(i \in \text{inds}\ sequence) \cdot sequence[i] = s$	The unique element of *sequence* which equals s

Environment

$E =$ **Env**(_,_,_,_,_,_,_,_)	Transformation Environment
E.fieldname	field selection in the **Kernel**
(*E.filedname*)[*number*]	indexing

Kernel

typedec(_,_)	**Kernel** data structure with wild-card entries
TypeOpt	Type structure with optional element nil
TypeSeq	Non-empty sequence of *Types*
kType	'Type' belonging in the **Kernel**

Intentionally Blank

B Kernel of ELLA Data Structure

B.1 Conventions

abc	$\in$	Abc (ie. it is an element of the set Abc)
Indexer, Size, Fnno	$\subseteq$	$\mathbb{N}_1$
Typeno, Tagno, Inputno	$\subseteq$	$\mathbb{N}_1$
Signalno, Delaytime	$\subseteq$	$\mathbb{N}_1$
Interval, Ambigtime	$\subseteq$	$\mathbb{N}$
Skew	$\subseteq$	$\mathbb{Z}$
Inputtype, Outputtype	$\subseteq$	Type
Initialvalue, Ambigvalue	$\subseteq$	Const
Fnname, Biopname	$\subseteq$	Upper case identifier or operator
Name, Signalname	$\subseteq$	Lower case identifier
Typename, Tagname	$\subseteq$	Lower case identifier
Lowerbound, Upperbound	$\subseteq$	positive or negative integer
Character	$\subseteq$	printable character

B.2 Data Structures

Enumerated Type Values

Enumerated	::=	Enum
	|	**string**(Typeno × TagnoSeq)
Enum	::=	**enum**(Typeno × Tagno)

Signal Types

Type	::=	**typeno**(Typeno)
	|	**typename**(Typename × Type)
	|	**stringtype**(Size × Type)
	|	**types**(TypeSeq)
	|	**typevoid**

Constants (Initialisation parameters)

Const	::=	Enumerated
	|	**conststring**(Size × Const)
	|	**consts**(ConstSeq)
	|	**constassoc**(Enum × Const)
	|	**constquery**(Type)
	|	**constvoid**

Constant Sets (Case Clause chooser values)

Constset ::= Enumerated
| **constsetalts**(ConstsetSeq)
| **constsetstring**(Size × Constset)
| **constsets**(ConstsetSeq)
| **constsetassoc**(Enum × Constset)
| **constsetany**(Type)

Units (Value delivering expressions)

Unit ::= Enumerated
| **conc**(Unit × Unit × Outputtype)
| **unitstring**(Size × Unit)
| **units**(UnitSeq)
| **instance**(Fnno × Unit)
| **unitassoc**(Enum × Unit)
| **extract**(Unit × Enum)
| **signal**(Signalno)
| **index**(Unit × Indexer × Outputtype)
| **trim**(Unit × Indexer × Indexer × Outputtype)
| **dyindex**(Unit × Unit × Outputtype)
| **replace**(Unit × Unit × Unit)
| **unitquery**(Type)
| **caseclause**(Unit × CaseSeq × Unit)
| **unitvoid**

Case ::= **case**(Constset × Unit)

Function Declarations

Fndec ::= **fndec**(Fnname × Inputtype × SignaldecSeq × Outputtype × Fnbody)

Signaldec ::= **signaldec**(Signalname × Type × Unitorinput)

Unitorinput ::= Unit
| **input**

Fnbody ::= Unit
| **reform**
| **biop**(Biopname)
| **delay**(Initialvalue × Ambigtime × Ambigvalue × Delaytime)
| **idelay**(Initialvalue × Delaytime)
| **sample**(Interval × Initialvalue × Skew)
| **ram**(Initialvalue)

Type Declarations

Typedec	::=	**typedec**(Typename × New)
New	::=	**tags**(TagSeq)
	|	**ellaint**(Tagname × Lowerbound × Upperbound)
	|	**chars**(Tagname × CharacterSeq)
Tag	::=	**tag**(Tagname × TypeOpt)

Environment Closure

Closure	::=	TypedecSeq × FndecSeq

Intentionally Blank

References

[Com90] Computer General Electronic Design, Greenways Business Park, Chippenham, Wiltshire, SN15 1BN, United Kingdom. *The ELLA Language Reference Manual.* 4.0th edition, 1990.

[Dav88] M. Davies. *Mathematical equivalence in a Primitive ELLA.* Internal Memorandum 4225, Royal Signals and Radar Establishment, Great Malvern, 1988.

[HM92] M.G. Hill and J.D. Morison *Executable Transformational Rules from Core ELLA to the Kernel.* Internal Memorandum 4629, Defence Research Agency, RSRE, Great Malvern, 1992.

[HWM90a] M.G. Hill, E.V. Whiting, and J.D. Morison. *Formal Semantic Definition of ELLA Timing.* Internal Memorandum 4436, Royal Signals and Radar Establishment, Great Malvern, 1990.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1990

[MH91] J.D. Morison and M.G. Hill *A Formal Definition of the Static Semantics of ELLAs Core.* Internal Report 91024, Royal Signals and Radar Establishment, Great Malvern, 1991.

[Tai88a] S. Tait. *System specification for a new simulator.* Internal Memorandum P209.40.4, Praxis, 1988.

[Tai88b] S. Tait. *Formal Specification of Built in Operations.* Internal Memorandum P209.40.5, Praxis, 1988.

INTENTIONALLY BLANK

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ........UNCLASSIFIED..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No. MEMO 4630	Month AUGUST	Year 1992

Originators Name and Location DRA, ST ANDREWS ROAD MALVERN, WORCS WR14 3PS
Monitoring Agency Name and Location
Title THE DYNAMIC SEMANTICS OF KERNEL ELLA

Report Security Classification UNCLASSIFIED	Title Classification (U, R, C or S) U

Foreign Language Title (in the case of translations)
Conference Details

Agency Reference	Contract Number and Period
Project Number	Other References
Authors HILL, M G	Pagination and Ref 49

Abstract

This document describes the dynamic semantics of the Kernel of ELLA. The Kernel is a set of data structures into which any ELLA circuit can be transformed. The semantics of two simple languages are explored in order to demonstrate the implementatability of the approach undertaken. Correspondence between this work and former analysis is shown.

Abstract Classification (U, R, C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

580/48

INTENTIONALLY BLANK